THE EXPERT'S VOICE® IN OPEN SOURCE

Covers
Python 3.1

Dive Into Python 3

All you need to know to get off the ground with Python 3

python

Mark Pilgrim

Foreword by Jesse Noller

Apress®

目錄

Dive Into Python3

版权信息

《深入 Python 3》 的内容涵盖了 Python 3 及其与 Python 2 的区别。相对《深入 Python》而言，它 20% 的内容进行了修订，80% 的内容是全新的。这本书现在已经完成了，但随时欢迎反馈意见。


```
you@localhost:~$ hg clone http://hg.diveintopython3.org/ diveintopython3
```

Chapter -1 《深入 Python 3》中有何新内容

> " 这不正是我们进来的地方吗？ " — 《迷墙》

又叫做 “the minus level”

你读过原版的 “深入 Python” 并可能甚至买了纸版的。（谢谢！）你差不多已经了解 Python 2 了。你准备好了投入到 Python 3 里面。… 如果所有这些都成立，继续读。（如果没有一个是成立的，你最好从头开始。）

Python 3 提供了一个脚本叫做 `2to3` 。学习它。喜欢它。使用它。用 `2to3` 移植代码到 Python 3 是一个有关 `2to3` 工具能够自动整理的所有东西的参考手册。很多这些东西都是语法的变更，因此了解 Python 3 里面许多的语法变更是一个好的起点。（ `print` 现在是一个函数， `x` 不能使用，等等。）

案例分析：移植 `chardet` 到 Python 3 记录了我努力（最终成功）把一个不平常的库从 Python 2 移植到 Python 3 的过程。它也许能帮助你；也许不能。这里存在一个相当陡的学习曲线，由于你首先需要稍微理解一下这个库，那样你才可以理解为什么它会损坏以及我如何修复它的。围绕字符串有很多损坏的地方。说到这个…

字符串。哈。从哪儿开始呢。Python 2 有 “strings” 和 “Unicode strings”。Python 3 有 “bytes” 和 “strings”。也就是说，现在所有字符串都是 Unicode 的字符串，那么如果你想处理一个字节包，你可以使用新的 `bytes` 类型。Python 3 从不会在 strings 和 bytes 之间进行隐式的转换，因此在任何时候如果你不确信你拥有的是什么类型，你的代码几乎无疑的将会出问题。阅读 Strings 的章节 了解更多细节信息。

贯穿整个这本书，Bytes 和 strings 的对比会一次又一次的出现。

- 在文件这章，你将了解到通过“二进制”模式和“文本”模式读取文件的区别；在文本模式下读取（和写入！）文件需要提供一个 `encoding` 参数。一些文本文件方法按照字符来计数，而另一些方法按照字节计数。如果你的代码采取一个字符等于一个字节的方式，那么在多字节表示一个字符的情况下将会出问题。
- 在 HTTP Web 服务这章， `httplib2` 模块通过 HTTP 获取头信息和数据。HTTP 头信息返回的是字符串，而 HTTP 正文则返回的是字节。
- 在序列化 Python 对象这章，你将了解到为什么 Python 3 里面的 `pickle` 模块定义了一个和 Python 2 向后不兼容的新的数据类型。（提示：这就是因为字节和字符串的原因。） 同样 JSON 也根本不支持字节类型。我将向你展示如何解决这个问题。
- 在案例分析：移植 `chardet` 到 Python 3这章，到处都是一大堆一大堆关于字节和字符串的东西。

即使你不关心 Unicode （但实际上你会的），你也会想阅读一下 Python 3 里面的字符串格式，这和 Python 2 里面的完全不一样。

迭代在 Python 3 里面无处不在，比起五年之前我写“深入Python” 的时候，我现在能更好的理解它们。你也需要理解他们，因为过去经常在 Python 2 里面返回列表的很多函数，在 Python 3 里面将返回迭代。至少，你应该阅读一下迭代章节的下半部分和高级迭代章节的下半部分。

根据大家的要求，我已经添加了一个关于特殊方法名称的附录，有点像 Python 文档的 “数据模型”章节但是包含更多的内容。

当我在撰写“深入 Python”的时候，所有可用的 XML 库都很糟糕。接着 Fredrik Lundh 编写了非常优秀的 ElementTree。Python 的专家们聪明的把 ElementTree 变成了标准库的一部分，然后现在它构成了我的新的 XML 章节的基础。解析 XML 的那些老的方式仍然可用，但是你应该避免使用它们，因为他们很糟糕！

除此之外，还有个关于 Python 的新东西 — 不是语言上的，而是社区中的 — 像 Python 包装索引(PyPI)的出现。Python 提供了实用工具类用来将你的代码打包成标准格式，并分发那些包到 PyPI 中。阅读 打包 Python 库了解详细信息。

Chapter 0 安装 Python

> " *Tempora mutantur nos et mutamur in illis.* （时光流转，吾等亦随之而变。） " — 古罗马谚语

深入

欢迎来到 Python 3 的世界。让我们继续深入。本章中，您将安装适合自己的 Python 3 版本。

何种版本的 **Python** 适合您？

对 Python 要做的第一件事情是安装。还是说已经装了？

如果使用的是托管服务器上的帐号， ISP［互联网供应商］ 可能已经安装了 Python 3 。如果是在家运行的 Linux ，也可能已经安装了 Python 3 。多数流行的 GNU/Linux 发行包在缺省安装中都包括了 Python 2 ；为数不多但却不断增加的发行包中同时也包括了 Python 3 。Mac OS X 包括了命令行版本的 Python 2，但直至本书写作之时止，其尚未提供 Python 3。Microsoft Windows 未安装任何版本之 Python 。但是不要绝望！无论是何种操作系统，均可通过安装 Python 来开启通向光明的道路。

在 Linux 或 Mac OS X 系统上检测 Python 3 的最简单办法是进入命令行。在 Linux 中，可从 `Application［应用程序］` 菜单找到叫一个做 `Terminal［终端］` 的程序。（它也有可能位于像 `Accessories［附件］` 或 `System［系统］` 这样的子菜单内。） 在 Mac OS X 中，在 `/Application/Utilities/` 文件夹中有一个叫做 `Terminal.app` 的应用程序。

见到命令行提示符之后，只需输入 `python3` （全部字母小写、无空格），并观察接下来发生的事情。我家中的 Linux 系统已经安装了 Python 3 ，运行该命令将把我带入*Python* 交互式 *shell* 中。

```
mark@atlantis:~$ python3
Python 3.0.1+ (r301:69556, Apr 15 2009, 17:25:52)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

（输入 `exit()` 并按下 `回车键` 可退出 Python 交互式 shell。）

我选择的 虚拟主机服务商 也运行 Linux 并提供命令行访问，但我的服务器未安装 Python 3 。（嘘！）

```
mark@manganese:~$ python3
bash: python3: command not found
```

因此无论在计算机上已安装了哪个版本，让我们回到本节开始时提到的问题：“哪种 Python 版本适合你？“，

［阅读关于 Windows 的指导，或者跳到 在 Mac OS X 上安装、 在 Ubuntu Linux 上安装 或 在其它平台上安装。］

在 Microsoft Windows 上安装

当前 Windows 有两种架构： 32位 和 64位。当然，还有很多不同的 Windows 版本 — XP、Vista、 Windows 7 — 而 Python 可在所有这些版本上运行。The more important distinction is 32-bit v. 64-bit. 如果不知道目前正在运行何种架构，那么多半是 32 位的。

访问 `python.org/download/` 并下载与计算机架构对应的 Python 3 Windows 安装程序。面对的选择可能包括下面这些：

- **Python 3.1 Windows** 安装程序 （Windows 二进制 — 不包括源码）
- **Python 3.1 Windows AMD64** 安装程序 （Windows AMD64 二进制 — 不包括源码）

未在此处提供直接下载链接是因为 Python 总是在进行小的更新，而我又不想为您错过更新负责。应该总是安装最新的 Python 3.x 版本，除非您有特别的理由不这么做。

1. 

   下载完成后，双击该 `.msi` 文件。由于正要运行的是可执行代码，Windows 将弹出一个安全警告。官方 Python 安装程序由负责 Python 开发的非盈利性组织 Python 软件基金会 进行数字签名。千万别接受山寨版！

   点击 `Run [运行]` 按钮启动 Python 3 安装程序。

2. 

安装程序将会询问的第一个问题是：是为所有用户，还是仅为您自己安装 Python 3。缺省的选项是 “为所有用户安装”，如果没有更好理由选择其它选项，这是最好的选择。（想要”只为我安装“的一个可能原因是：正往公司的计算机上安装 Python 而您的 Windows 帐号又没有 Administrator 权限。不过，您又为啥未经公司 Windows 管理员的许可而安装 Python 呢？这个问题上不要给我惹麻烦！）

点击 `Next [下一步]` 按钮接受对安装类型的选择。

3. 

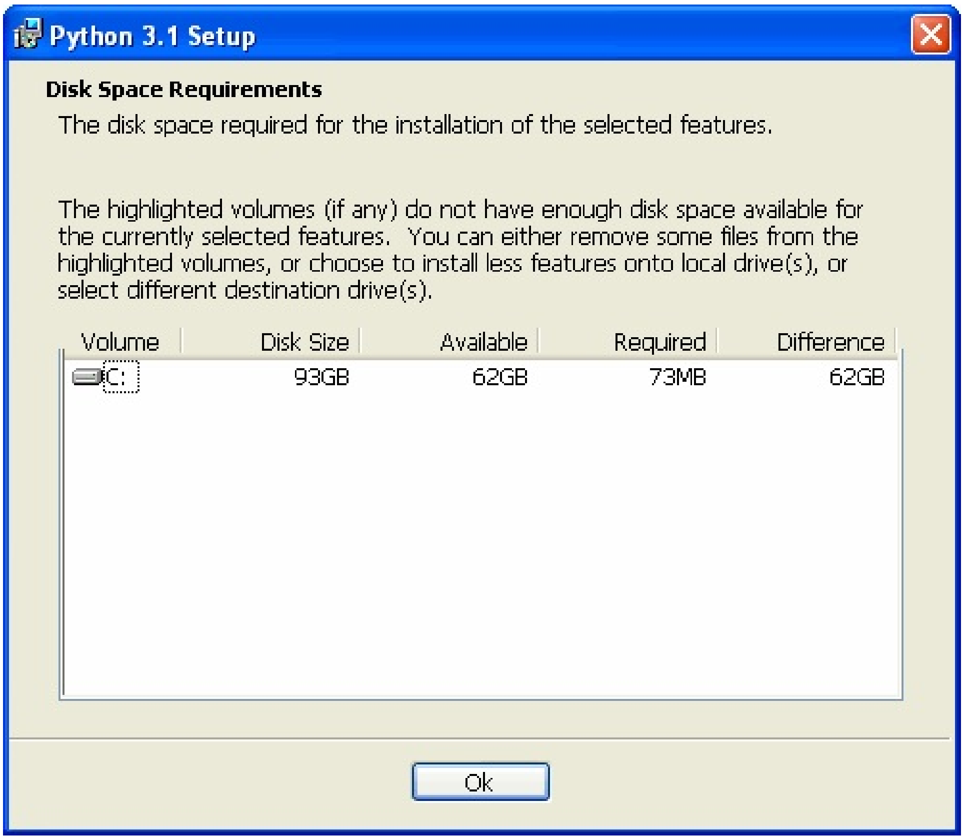
接下来，安装程序将会提示选择一个目标目录。所有 Python 3.1.x 版本缺省的目标目录是： `C:\Python31\` ，这对绝大多数用户都是合适的，除非您有特别的理由修改它。如果有单独的磁盘驱动器用于安装应用程序，可通过嵌入式控件找到它，或直接在下方的文本框中输入该路径名。如果在 `C:` 盘安装 Python 受限；可在其它盘的任何目录下安装。

点击 `Next [下一步]` 按钮接受对目标目录的选择。

4. 

接下来的页面看着有点复杂，但其实并不真的复杂。和其它安装程序一样，您可以选择不安装 Python 3 每个单独部件。如果磁盘空间特别紧张，可以将某些部件排除在外。

- **Register Extensions** ［注册扩展名］ 允许通过双击 Python 脚本 (`.py` files) 来运行它们。建议选上，但不是必需的。（该选项不占用任何磁盘空间，因此排除它没有任何意义。）
- **Tcl/Tk** 是 Python Shell 使用的图形化类库，您将在整本书都用到它。强烈建议保留该选项。
- **Documentation** ［文档］ 安装的帮助文件包括大量来自 `docs.python.org` 信息。如果使用拨号上网或者互联网访问受限的话，建议保留。
- **Utility Scripts** ［实用脚本］ 包括本书稍后将学到的 `2to3.py` 脚本。如果想学习如何将现有 Python 2 代码移植到 Python 3 ，这是必需的部件。若无现有的 Python 2 代码，可略过该选项。
- **Test Suite** ［测试套件］ 是用于测试 Python 解释器的脚本集合。本书中将不会用到，而且我在用 Python 编程的过程中也从未用到。完全是可选的。

5. 

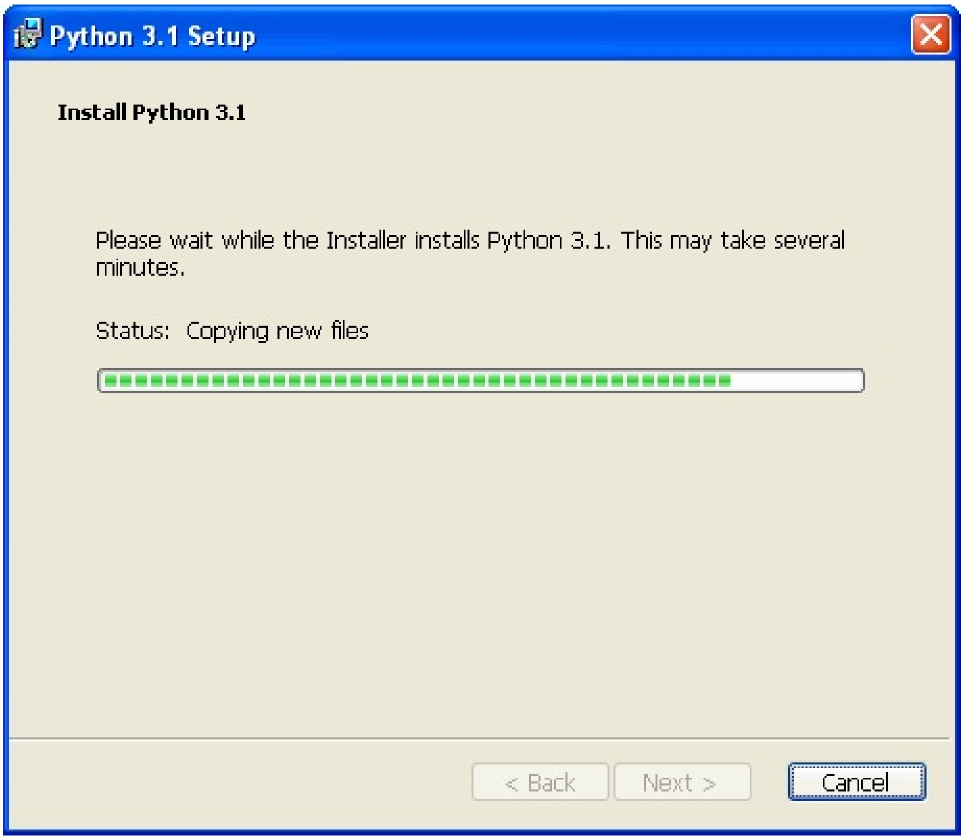
如果不确定有多少磁盘空间，点击 `Disk Usage [磁盘使用情况]` 按钮。安装程序将列出所有驱动器盘符，并计算每个驱动器上有多少可用空间，以及安装后会剩下多少空间。

点击 `OK [确定]` 按钮返回“Customizing Python［自定义 Python］”页面。

6. 

如果决心排除某选项，选择选项之前的下拉选项按钮并选中 “Entire feature will be unavailable.［整个功能将不可用］”选项。例如，排除 Test Suite ［测试套件］将节省高达 7908KB 的磁盘空间。

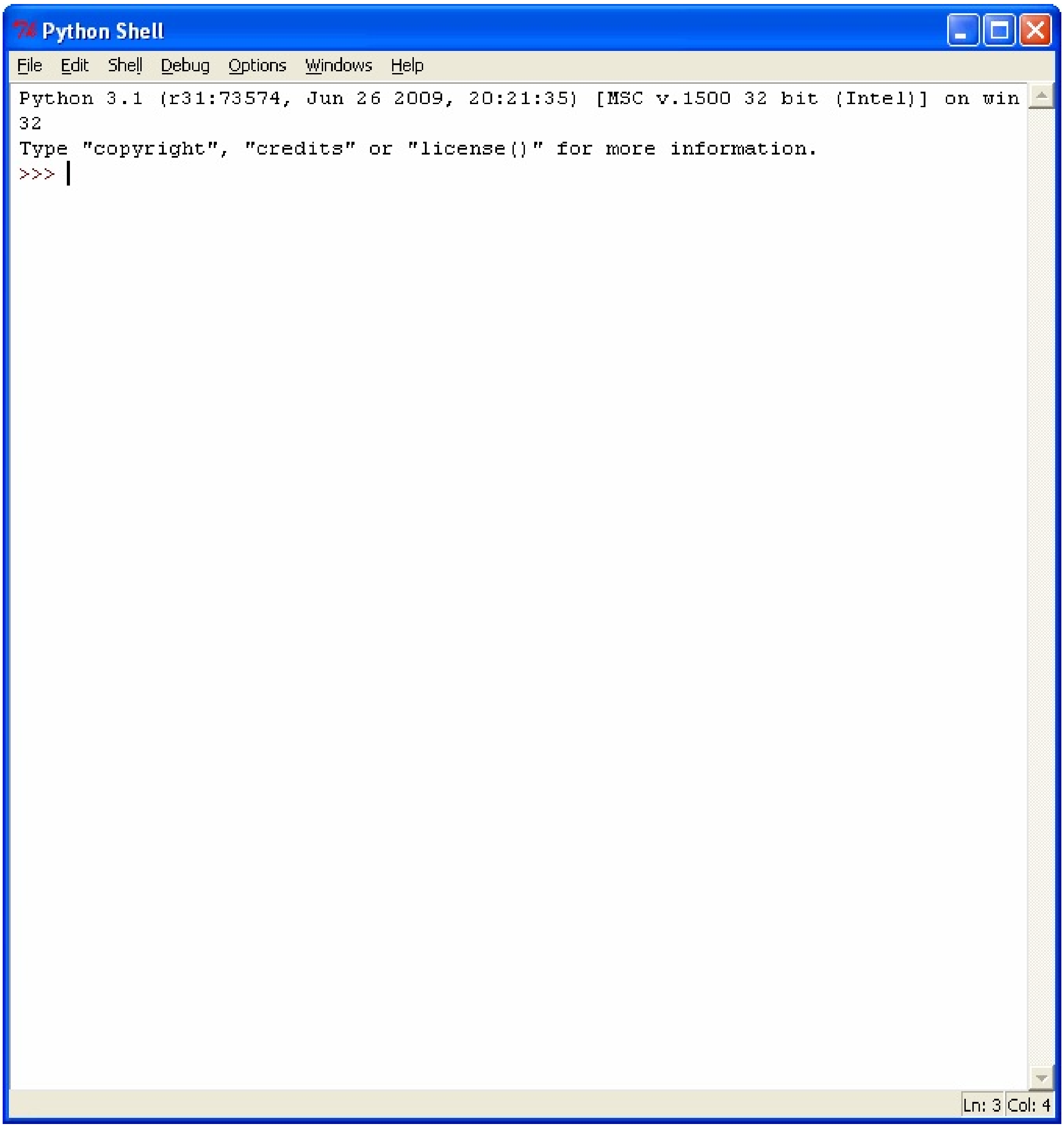
点击 `Next［下一步］` 按钮接受对所选内容的选择。

7. 

安装程序将把所有必需的文件拷贝到所选择的目标目录中。（该过程非常快捷，以至于我不得不试了三遍才捕捉到它的屏幕截图！）

8. 

点击 `Finish [完成]` 按钮退出该安装程序。

9. 

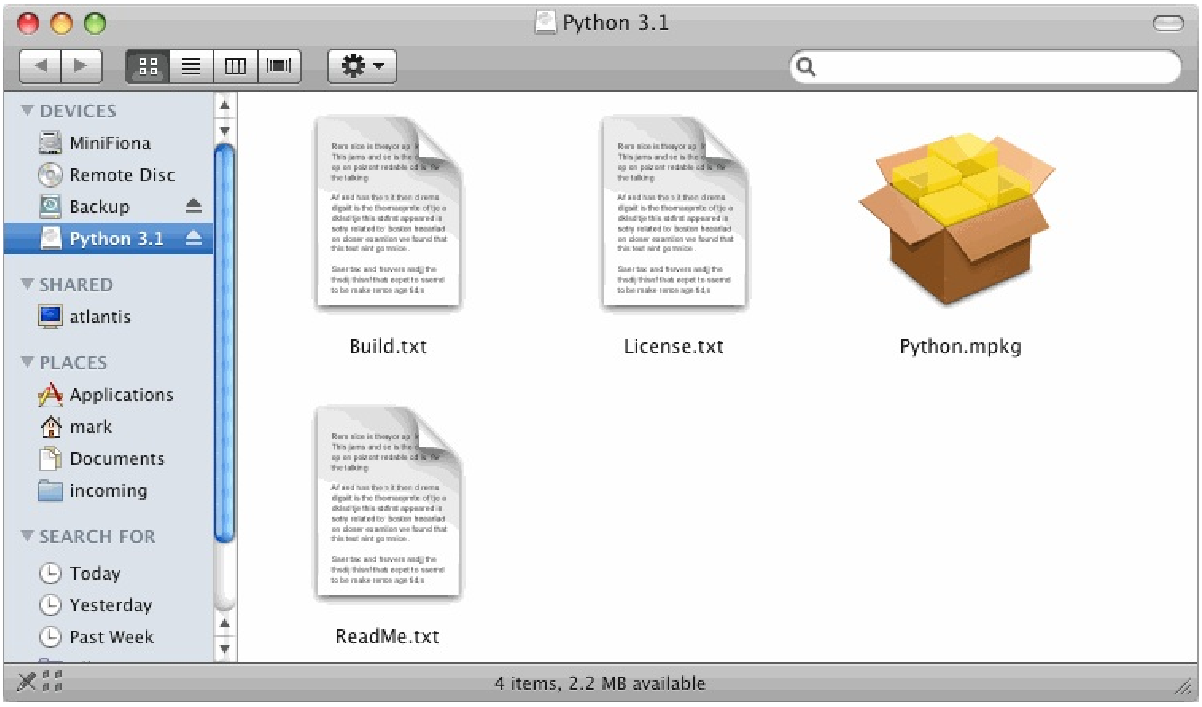
在 `开始` 菜单中，将会出现一条名为 `Python 3.1` 的新菜单项。在其中有一个名为 IDLE 的程序。选择此菜单项以运行交互式 Python Shell 。

[跳到 使用 Python Shell]

在 Mac OS X 上安装

所有的现代麦金塔计算机使用英特尔芯片（像大多数 Windows PC 一样)。旧款的苹果电脑使用 PowerPC 芯片。你无须理解其中区别，因为所有苹果电脑只有一种 Mac Python 安装程序。

访问 `python.org/download/` 并下载 Mac 安装程序。它可能被叫做 **Python 3.1 Mac Installer Disk Image** 之类的名字，尽管版本号可能会不同。请确定下载的是 3.x 版，而不是 2.x 版。

1. 

浏览器可以自动挂载磁盘映像，并打开一个 Finder 窗口展示其内容。（如果没有发生这样的情形，则需要在下载目录中找到磁盘映像，并双击挂载。它可能被命名为 `python-3.1.dmg` 之类的名称。）磁盘映像包括一些文本文件（ `Build.txt` 、 `License.txt` 、 `ReadMe.txt` ），以及实际的安装程序包， `Python.mpkg` 。

双击 `Python.mpkg` 安装程序包以启动 Mac Python 安装程序。

2. 

安装程序的第一页就 Python 本身给出了一段简要描述，然后提示您参阅 `ReadMe.txt` 文件（您没有读过该文件，不是吗？）以掌握更多细节。

点击 `Continue [继续]` 按钮进入下一步。

3. ![Python 安装程序：所支持的架构、磁盘空间及可接受目标目录等相关信息]

   接下来的页面实际包含一些重要信息： Python 必须安装在 Mac OS X 10.3 或其后续版本之上。如果仍在使用 Mac OS X 10.2，那就真的需要升级一下了。苹果公司已经不再为（Mac OS X 10.2）操作系统提供安全更新了，而且如果曾经上网的话，您的计算机可能已经处于危险之中了。此外，您也无法运行 Python 3 。

   点击 `Continue [继续]` 按钮继续前进。

4. 

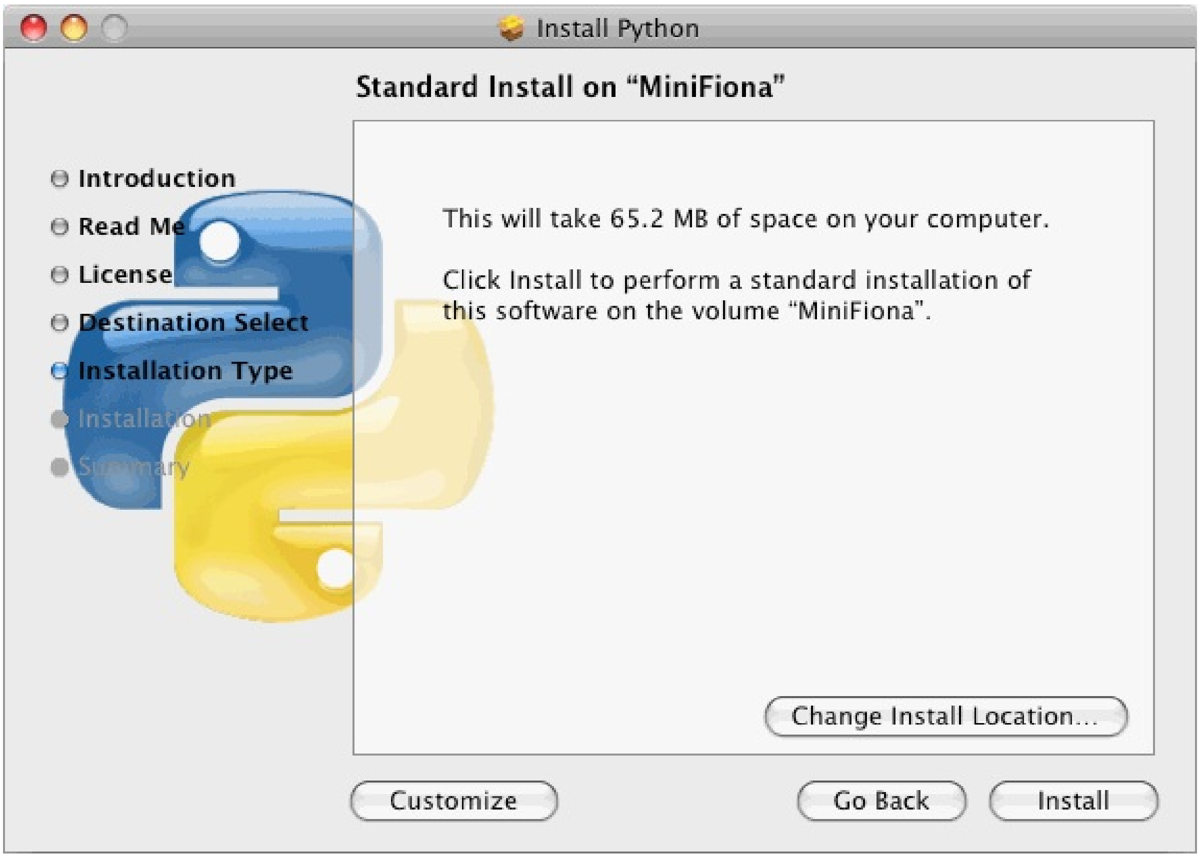
   如同所有优秀的安装程序，Python 安装程序列出了软件许可协议。Python 是开源软件，其许可协议由 Open Source Initiative［开源软件促进会］ 提供。历史上，Python 有过一些所有者和赞助者，每个都在软件许可协议之上留下了痕迹。但最终结果是：Python 是开源的，可在任何平台上为任何目的使用它，而无需付费或承担对等义务。

   再次点击 `Continue [继续]` 按钮。

5. 

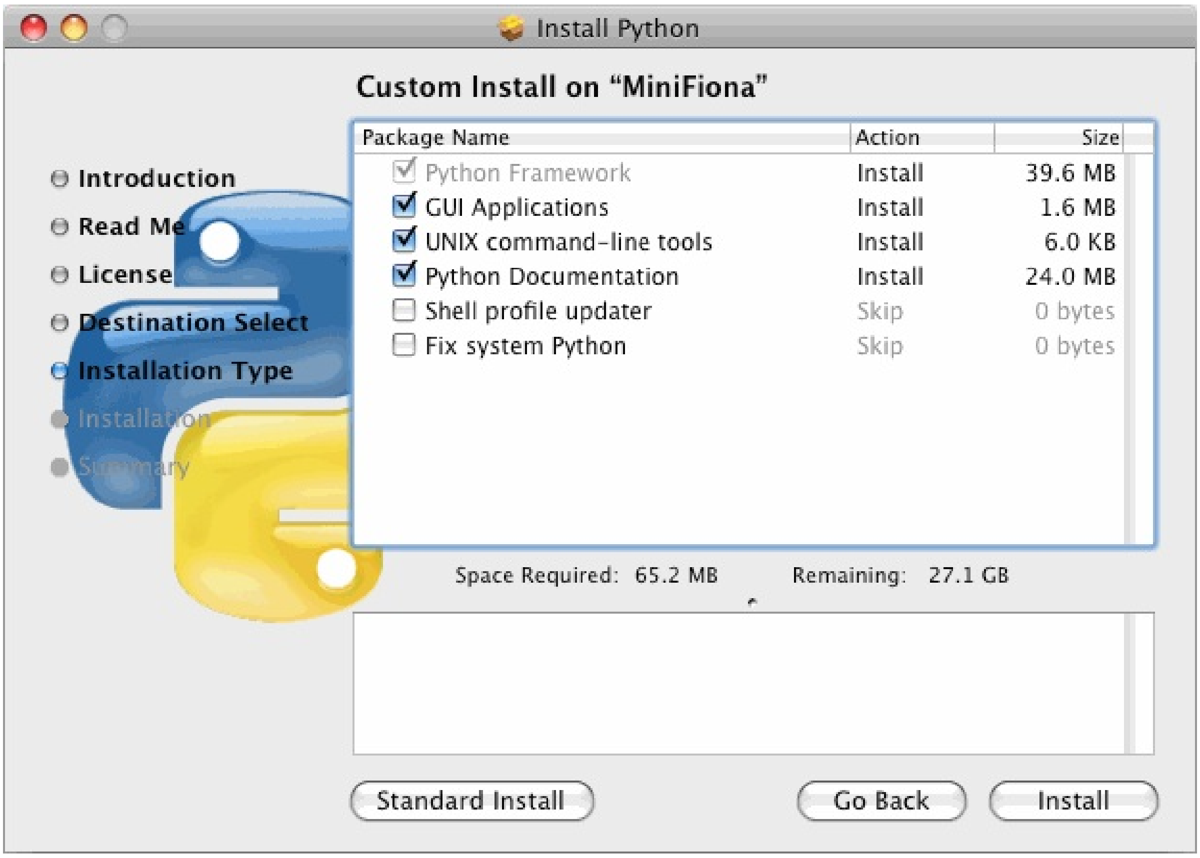
根据苹果安装程序框架的习惯，必须“agree［同意］”软件许可协议以完成安装。由于 Python 是开源的，实际上您所“同意”的只是授予您额外的权利，而不是剥夺它们。

点击 `Agree［同意］` 按钮以继续安装。

6. 

下一个画面允许您修改安装位置。必须 将 Python 安装到启动驱动器上，但由于安装程序的限制，它并没有强迫这么做。说实话，我从来没有需要过修改安装位置。

从该画面中，您还可以自定义安装以剔除特定功能。如果想这么做，点击 `Customize［自定义］` 按钮；否则点击 `Install［安装］` 按钮。

7. 

如果选择了自定义安装，安装程序将为您提供下列功能：

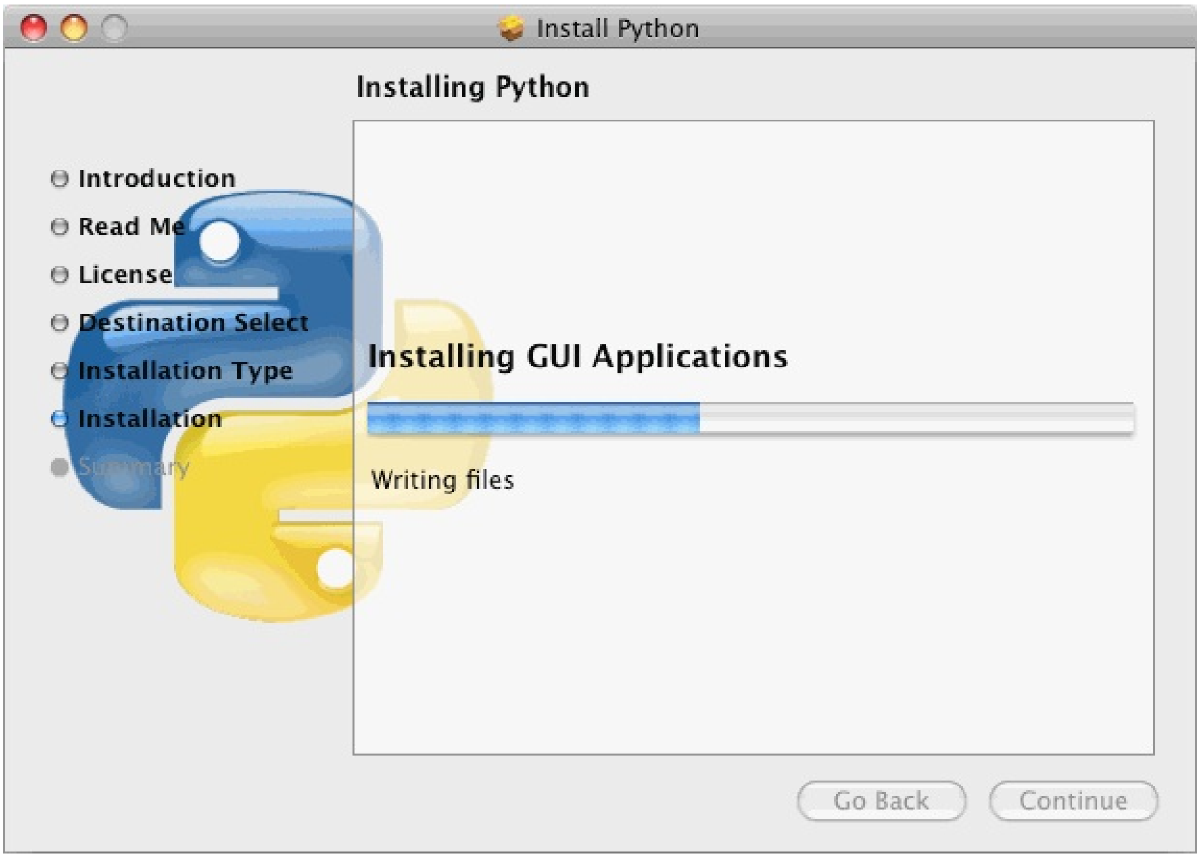
- **Python Framework**［**Python** 框架］. 这是 Python 的核心所在，由于必须被安装，它已经被选中并处于无法取消状态。
- **GUI Applications**［**GUI** 应用程序］ 包括 IDLE，即本书通篇将用到的图形化 Python Shell 。强烈建议保留该选项。
- **UNIX command-line tools**［**UNIX** 命令行工具］ 包括了 `python3` 命令行应用程序。同样强烈建议保留该选项。
- **Python Documentation**［**Python** 文档］ 包含了来自 `docs.python.org` 的许多信息。如果使用拨号上网或者互联网访问受限的话，建议保留。
- **Shell profile updater**［**Shell** 文档更新程序］ 控制是否更新 shell 设置（用于 `Terminal.app` 中）以确保此版本的 Python 位于 Shell 的搜索路径当中。您可能不需要修改该项设置。
- **Fix system Python**［修复系统 **Python**］ 不应作变更。（它告诉 Mac 将 Python 3 用作所有脚本的缺省 Python ，包括来自苹果公司的内置系统脚本。这将会导致非常糟糕的结果，因为多数这些脚本是为 Python 2 编写的，在 Python 3 环境中将无法正确运行。）

点击 `Install [安装]` 按钮以继续。

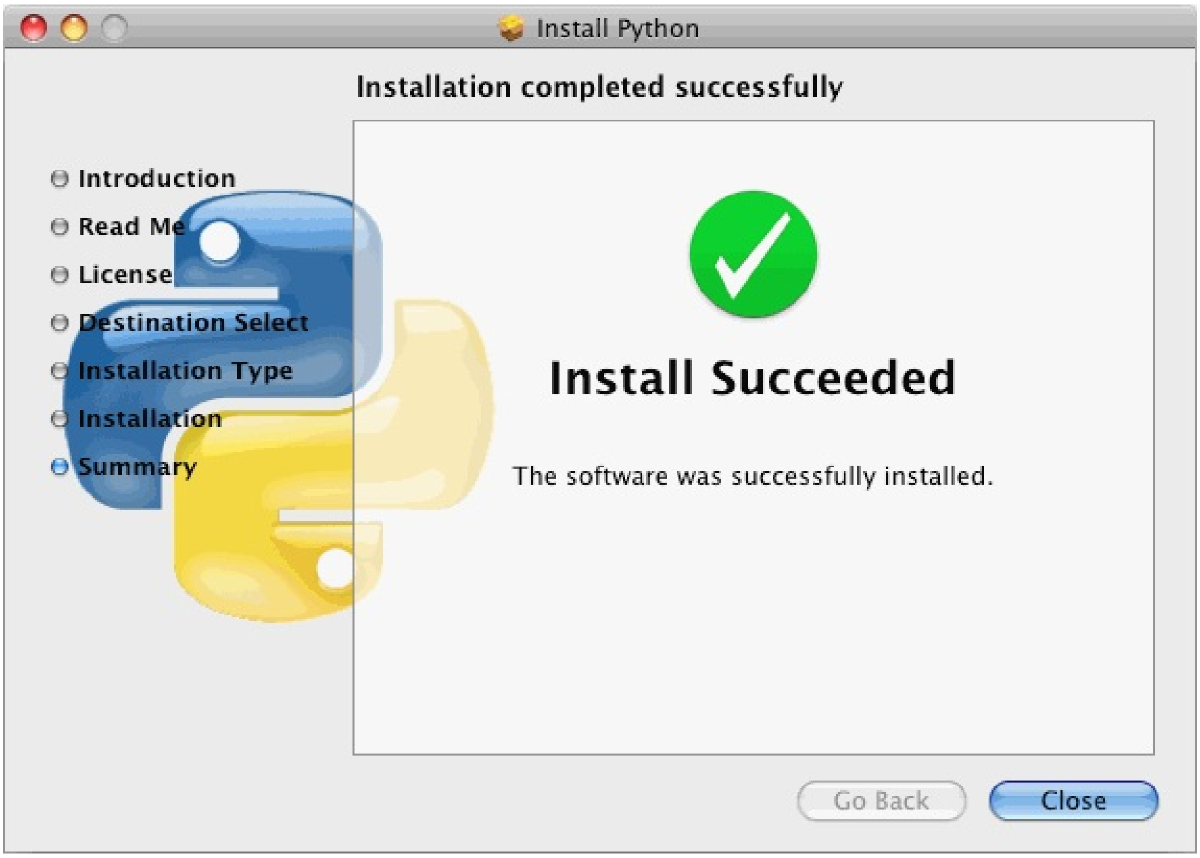
8. 

由于是安装系统级的框架，且二进制文件被安装至 `/usr/local/bin/` 之中，安装程序将会向您询问管理员口令。没有管理员权限是无法安装 Mac Python 的。

点击 `OK [确定]` 按钮开始安装。

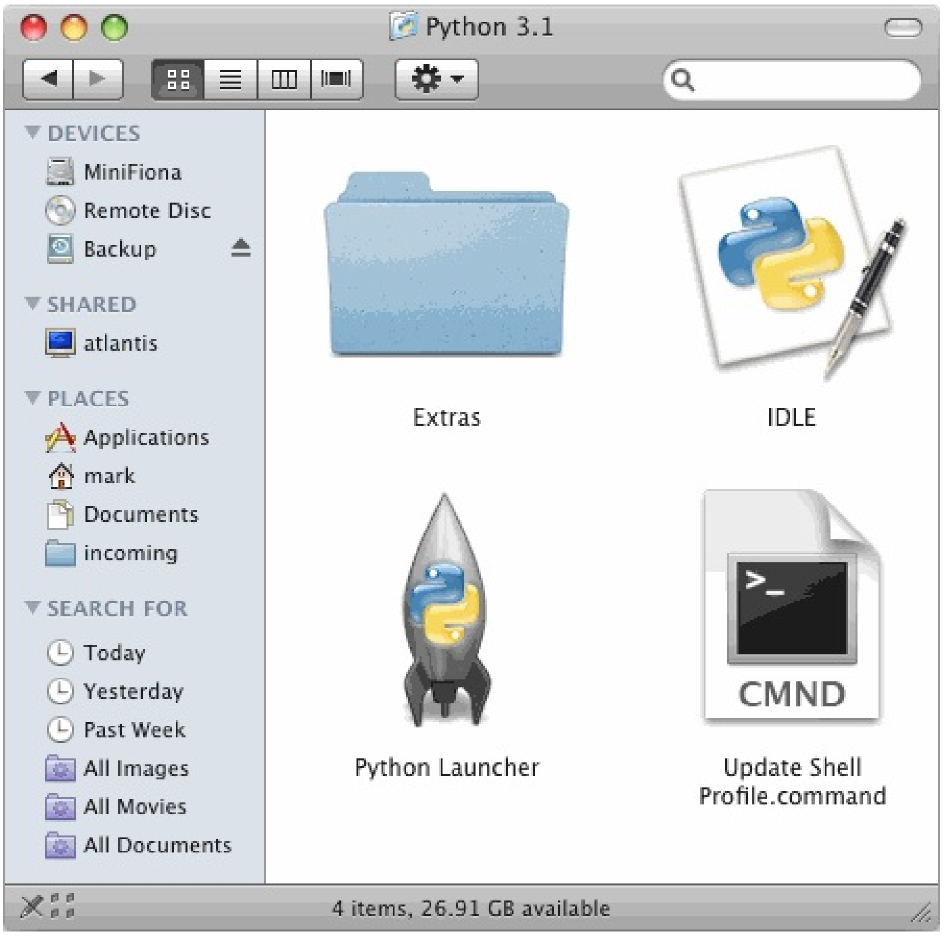
9. 

在安装所选功能时，安装程序将会显示进度条。

10. 

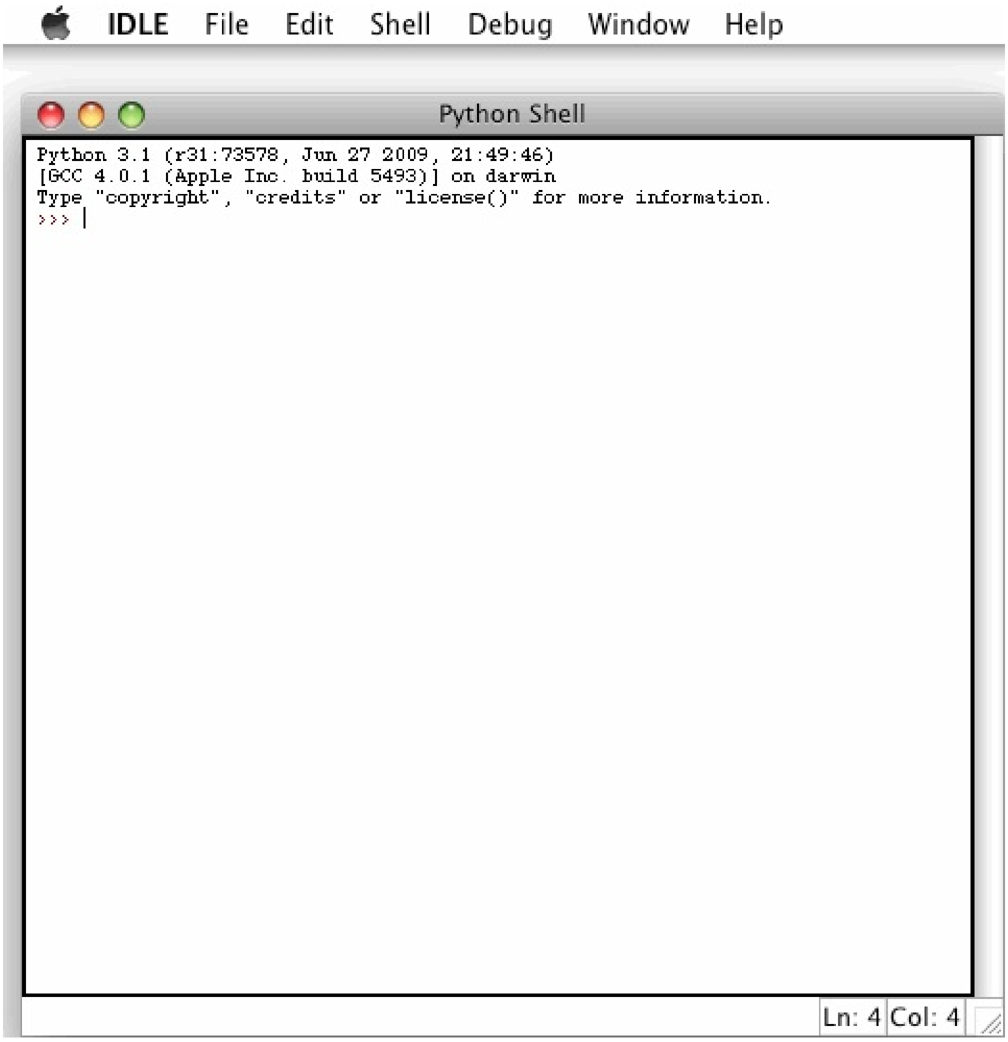
假定一切顺利，安装程序将会展示一个很大的绿色对号，告知安装成功完成。

点击 `Close [关闭]` 按钮退出该安装程序。

11. 

加入没有修改安装位置，您可以在 `/Applications` 目录下的 `Python 3.1` 目录中找到新安装的文件。 最重要的部分是图形化 Python Shell IDLE。

双击 IDLE 以启动 Python Shell。

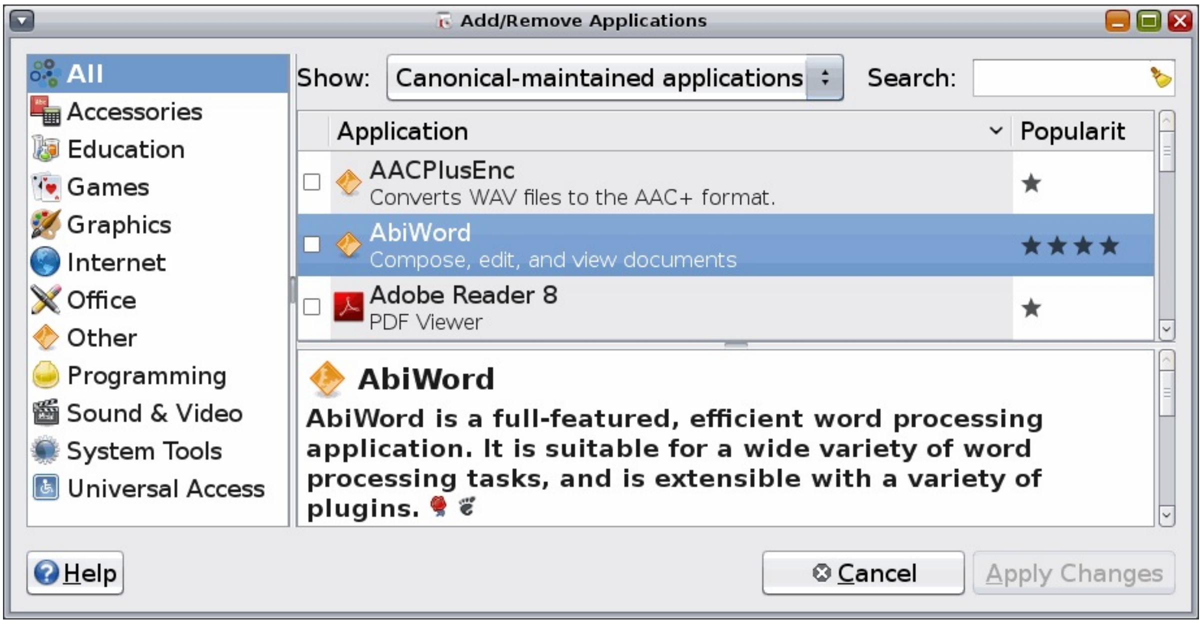
12. 

    Python Shell 是您探索 Python 过程中花费时间最多的地方。本书中所有的例子都假定您能够找到进入 Python Shell 的方法。

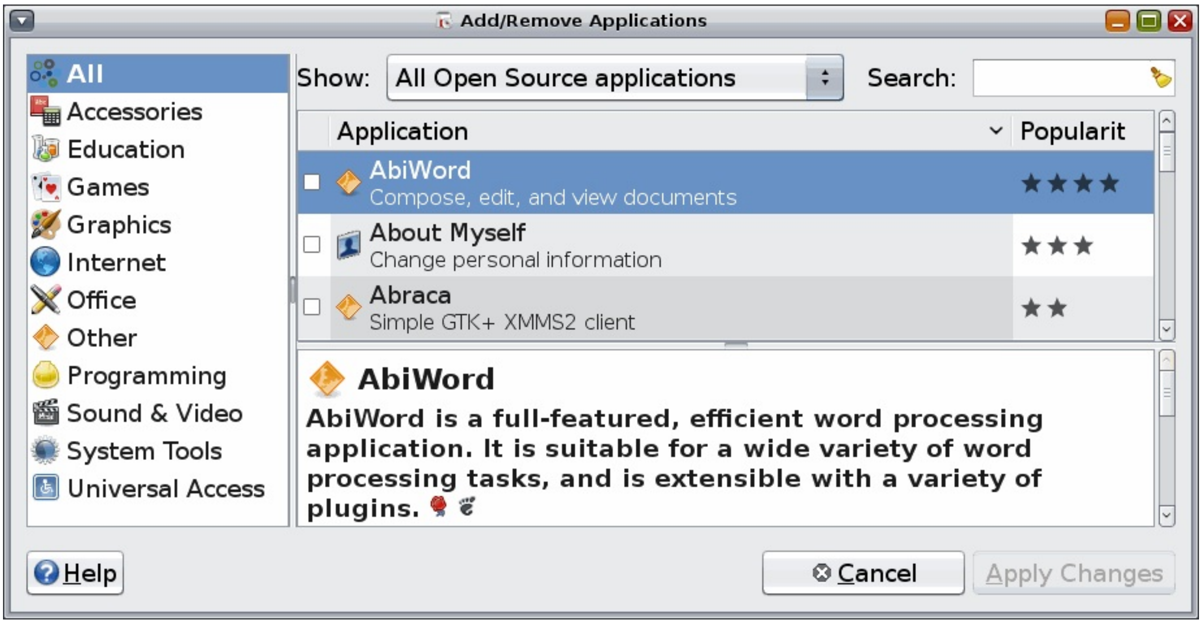
[跳到 使用 Python Shell]

在 Ubuntu Linux 上安装

现代的 Linux 发行版背后都有着大型的预编译应用程序仓库，随时可用于安装。具体的细节各发行版均不同。对于 Ubuntu Linux 而言，安装 Python 3 的最简单途径是通过 `Applications` 菜单中的 `增加／删除` 应用程序。

1. 

在首次运行 `增加／删除` 应用程序时，它将展示一份分成多类的预选程序清单。有的已经安装；多数还没有。因为该仓库包括超过 10,000 种应用程序，所以可以使用过滤器参看仓库的不同部分。默认过滤器是“由 Canonical 维护的应用程序”，它是创建及维护 Ubuntu Linux 的 Canonical 公司官方所支持的大量应用程序中的一个小子集。

2. 

Python 3 并非由 Canonical 维护，因此第一个步骤是下拉过滤器菜单，并选择“所有开源应用程序”。

3. 

放宽过滤器以包括所有开源应用程序之后，使用进紧挨着过滤器菜单的"搜索"框来搜索 `Python 3` 。

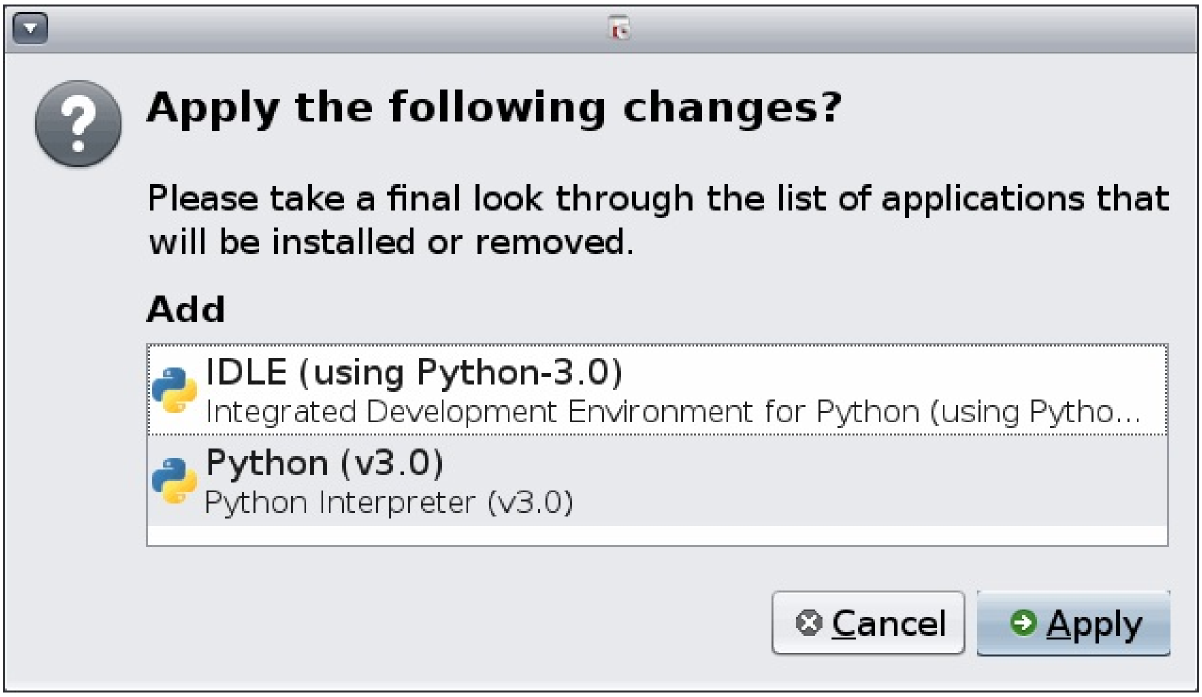
4. 

现在应用程序列表收窄为仅包括匹配 `Python 3` 的那些内容。您将查看两个安装包。第一个是 `Python (v3.0)` 。该安装包包含了 Python 解释器自身。

5. 

第二个要安装的包就在正上方： `IDLE (using Python-3.0)` 。这是你在整本书都要用到的图形化 Python Shell 。

选好这两个包后，点击 `Apply Changes [应用修改]` 按钮以继续。

6. 

该软件包管理器将会要求您确认是否要添加 `IDLE (using Python-3.0)` 和 `Python (v3.0)` 。

点击 `Apply [应用]` 按钮以继续。

7. 

在从 Canonical 互联网仓库下载所需安装包时，软件包管理器将显示一个进度条。

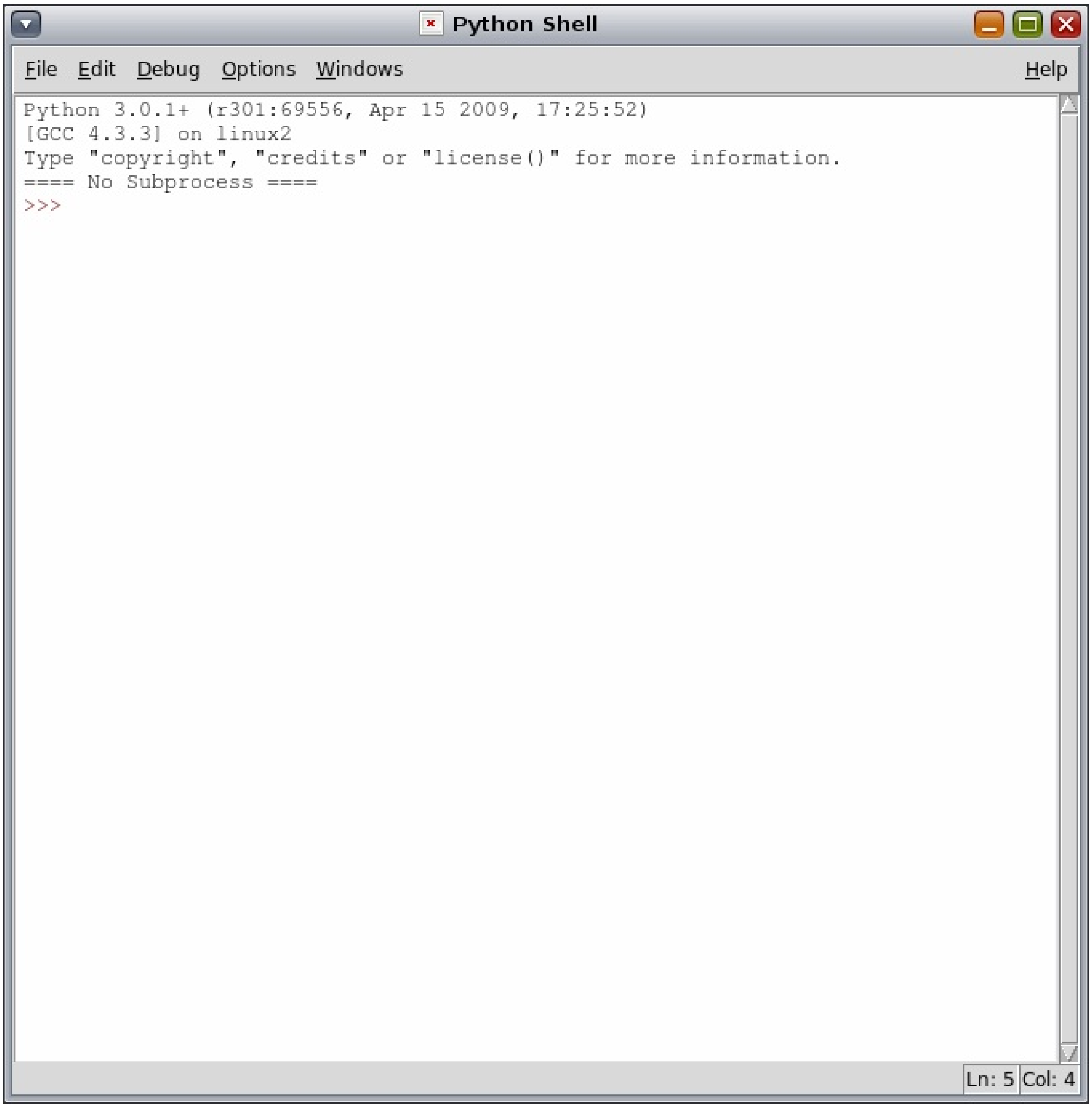
8. 

下好安装包后，软件包管理器将会自动开始安装。

9. 

如果一切顺利，软件包管理器将确认两个安装包都已安装成功。从此，您可双击 IDLE 启动 Python Shell，或者点击 `Close [关闭]` 按钮退出软件包管理器。

您还可以从 `Applications [应用程序]` 菜单，然后进入 `Programming` 子菜单并选择 IDLE，以重新启动 Python Shell。

10. 

    Python Shell 是您探索 Python 过程中花费时间最多的地方。本书中所有的例子都假定您能够找到进入 Python Shell 的方法。

[跳到 使用 Python Shell]

在其它平台上安装

Python 3 还可在一些其它平台上安装。特别要指出的是，它几乎可以在所有的 Linux、 BSD 和基于 Solaris 的发行版纸上安装。例如，RedHat Linux 使用 `yum` 软件包管理器；FreeBSD 有 移植和软件包集合；Solaris 有 `pkgadd` 和 friends 。在网上快速搜索 `Python 3` + 您的操作系统 将会告诉你是否存在该平台的 Python 以及如何安装。

使用 **Python Shell**

Python Shell 是您探索 Python 语法，通过命令获取交互式帮助以及调试段程序的地方。图形化 Python Shell （名为 IDLE）还包括了一个不错的文本编辑器，它支持 Python 语法着色并与 Python Shell 进行了整和。如果还没有喜欢的文本编辑器，不妨试用下 IDLE 。

重中之重。Python Shell 本身是一款了不起的互动环境。在本书中，您将看到下面这样的例子：

```
>>> 1 + 1
2
```

这三个尖括号， `>>>` ，表示 Python Shell 提示符。不要输入该部分。它只是让您知道该例要在 Python Shell 中运行。

`1 + 1` 是您输入的部分。您可在 Python Shell 中输入任何有效的 Python 表达式和命令。别怕羞，它不会咬你！最糟糕的事情也不过看到一条错误信息。命令将立即得到执行（一旦您按下 `ENTER [回车键]` ）；表达式的值将立即得到计算，而 Python Shell 将输出结果。

`2` 是该表达式的计算结果。事实上， `1 + 1` 是一个有效的 Python 不等式。结果，当然，是 `2` 。

让我们尝试下另一个例子.

```
>>> print('Hello world!')
Hello world!
```

很简单，不是吗？但你在 Python shell 中可完成的工作要多得多。如果您被困住了——无法想起某个命令，或者无法想起如何正确给某个函数传递参数——您可寻求 Python Shell 的交互式帮助。只需输入 `help` 并按下 `回车键` 。

```
>>> help
Type help() for interactive help, or help(object) for help about object.
```

有两种帮助模式。您可以获得某个对象的帮助，这样将只打印出文档并返回 Python Shell 提示符。您也可以输入 *help mode*，系统将不会计算 Python 表达式，您只需输入关键字或命令名称，系统将会输出关于该命令它所知道的内容。

要进入交互帮助模式，仅需输入 `help()` 并按下 `回车键` 。

```
>>> help()
Welcome to Python 3.0!This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".
help>
```

请注意提示符是如何从 `>>>` 改变为 `help>` 的。该提示符提醒您目前正处于交互式帮助模式。现在您可以输入任何关键字、命令、模块名称、函数名称 — 几乎任何 Python 能够理解的一切 — 然后阅读其文档。

```

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.

no Python documentation found for 'PapayaWhip'

 You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

1. 要获取 `print()` 函数的文档，仅需输入 `print` 然后按下 `回车键` 。该交互式帮助模式将会显示类似 man 页面的内容：函数名称、简要内容、函数的参数及缺省值等等。如果文档看起来很难懂，千万别慌。您将在后面不远的章节中学到关于这些概念的更多内容。
2. 当然，交互式帮助模式并不知道一切。如果您所输入的不是 Python 的命令、模块、函数或者其它内建关键字，交互式帮助模式将只能耸耸虚拟的肩膀。
3. 要退出交互帮助模式，仅需输入 `quit()` 并按下 `回车键` 。
4. 提示符将变回 `>>>` 以提示您已经离开交互帮助模式，并返回到了 Python Shell 。

图形化的 Python Shell —— IDLE,同样带有一个 Python 相关的文本编辑器。

Python 编辑器和集成开发环境

如果要以 Python 编写程序，IDLE 并不是唯一的编辑器选择。尽管它对于初学该语言非常有帮助，但许多开发人员更喜欢其它文本编辑器或集成开发环境。（IDEs）在此我不想展开阐述，Python 社区维护了一份 Python 相关编辑器的清单，涵盖了各种各样支持平台和软件许可协议。

您可能也想查看一下这份 Python相关 IDEs 的清单，尽管其中还只有少数才支持 Python 3 。其中之一是 PyDev，Eclipse 的一种插件，它将Eclipse 变成了一种成熟的 Python IDE。Eclipse 和 PyDev 都是跨平台的开源软件。

在商业方面，有 ActiveState 公司的 Komodo IDE 。它需要用户为单位的授权许可，但学生可以得到折扣，同时还有时间受限的免费试用版。

在用 Python 编程的九年中，我使用 GNU Emacs 编辑 Python 程序，并在命令行 Python Shell 中进行调试。对于使用 Python 开发来说，编辑器之选没有绝对的正确和错误。重要的是找到适合自己的道路！

Chapter 1 你的第一个 Python 程序

> " Don’t bury your burden in saintly silence. You have a problem? Great. Rejoice, dive in, and investigate. " — Ven. Henepola Gunaratana

Diving In

通常程序设计的书籍都会以一堆关于基础知识的章节开始，最终逐步的构建一些有用的东西。让我们跳过所有的那些东西，来看一个完整的、可以直接运行的 Python 程序。可能刚开始你根本看不懂，但不要担心，因为你会去一行一行的仔细研究。但是首先还是要通读一遍，看看里面什么东西（如果有的话）是你可以看懂的。

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
    if size < 0:
        raise ValueError('number must be non-negative')

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

现在让我们从命令行来运行这个程序。在 Windows 上，类似这样：

```
c:\home\diveintopython3\examples> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

在 Mac OS X 或者 Linux 上，类似这样：

```
you@localhost:~/diveintopython3/examples$ python3 humansize.py
1.0 TB
931.3 GiB
```

刚刚发生了什么？你执行了你的第一个 Python 程序。你从命令行调用了 Python 解释器，并且传递了一个你想 Python 去执行的脚本的名称。这个脚本定义了一个单一的函数，这个 `approximate_size()` 函数把一个精确到字节的文件大小计算成一个有漂亮格式（大约计算的）的大小。（你可能已经在 Windows Explorer，或者 Mac OS X Finder，或者 Linux 上的 Nautilus 或 Dolphin 或 Thunar 看到过这个。如果你按照多列的列表来显示一个文件夹的文档，它就会显示一个包含文档图标、文档名称、大小、类型、最后修改日期等等信息的表格。如果这个文件夹包含一个 1093 字节大小名叫 `TODO` 的文件，你的文件管理器将不会显示成 `TODO 1093 bytes` ，而用 `TODO 1 KB` 的显示格式代替。那就是 `approximate_size()` 函数所做的事情。）

看看这个脚本的底部，你会看到对 `print(approximate_size(` arguments `))` 的两次调用。这些叫做函数调用 —— 第一个调用了 `approximate_size()` 函数并传递了一些参数，接着直接把返回值传递给了 `print()` 函数。这个 `print()` 函数是内置的，你将从不会看到它的一个显式的声明。你只管在需要的任何时候任何地方使用它就行。（有很多内置函数，更多的函数独立于各个 *modules* （模块）里面。保持耐心，你会逐步熟悉它们的。）

那么为什么每次在命令行运行脚本都会给你同样的输出结果呢？我们将讲解这个。首先，让我们来看一下 `approximate_size()` 函数。

声明函数

像多数其他语言一样， Python 也有函数， 但是它没有像 C++ 一样的单独头文件，也没有像 Pascal 一样 `interface` / `implementation` （接口／实现）部分。当你需要一个函数的时候，就像这样声明它就行：

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

当你需要一个函数的时候，只要声明它就行。

函数声明以关键字 `def` 开头，紧跟着函数的名称，然后是用括号括起来的参数。多个参数以逗号分割。

同时注意，函数不定义一个返回数据类型。 Python 函数不指定它们的返回值的类型，甚至不指定它们是否返回一个值。（事实上，每个 Python 函数都返回一个值，如果这个函数曾经执行了 `return` 语句，它将返回那个值，否则它将返回 Python 里面的空值 `None` 。）

> ☞在某些语言里面，函数（返回一个值）以 `function` 开头，同时子程序（不返回值的）以 `sub` 开头。 Python 里面没有子程序。所有的东西都是一个函数，所有的函数都返回一个值（即使它是 `None` 值），并且所有的函数都以 `def` 开头。

`approximate_size()` 函数有两个参数 — `size` 和 `a_kilobyte_is_1024_bytes` — 但都没有指定数据类型。在 Python 里面，变量从来不会显式的指定类型。Python 会在内部算出一个变量的类型并进行跟踪。

> ☞在 Java 和其他静态类型的语言里面，你必须给函数返回值和每个函数参数指定数据类性。而在 Python 里面，你从来不需要给任何东西指定显式的数据类型。根据你赋的值，Python 会在内部对数据类型进行跟踪。

可选的和命名的参数

Python 允许函数函数有默认值。如果函数被调用的时候没有指定参数，那么参数将使用默认值。不仅如此，通过使用命名参数还可以按照任何顺序指定参数。

让我们再看一下 `approximate_size()` 函数的声明：

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

第二个参数 `a_kilobyte_is_1024_bytes` 指定了一个默认值 `True` 。 意思是这个参数是 *optional* （可选的），你可以在调用的时候不指定它，Python 将看成你调用的时候使用了 `True` 作为第二个参数。

现在看一下这个脚本的底部：

```
if __name__ == '__main__':
```

1. 这个对 `approximate_size()` 函数的调用指定了两个参数。在 `approximate_size()` 函数里面， `a_kilobyte_is_1024_bytes` 的值将为 `False` ，因为你显式的传入了 `False` 作为第二个参数。
2. 这个对 `approximate_size()` 函数的调用只指定了一个参数。但这是可以的，因为第二个参数是可选的！由于调用者没有指定，第二个参数就会使用在函数声明的时候定义的默认值 `True` 。

你也可以通过名称将值传入一个函数。

```
>>> from humansize import approximate_size

'4.0 KB'

'4.0 KB'

'4.0 KB'

  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg

  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

1. 这个对 `approximate_size()` 函数的调用给第一个参数（(`size` ）指定了值 `4000` ，并且给名为 `a_kilobyte_is_1024_bytes` 的参数指定了值 `False` 。（那碰巧是第二个参数，但这没有关系，马上你就会了解到。）

2. 这个对 `approximate_size()` 函数的调用给名为 `size` 参数指定了值 `4000`，并为名为 `a_kilobyte_is_1024_bytes` 的参数指定了值 `False`。（这些命名参数碰巧和函数声明时列出的参数顺序一样，但同样不要紧。）
3. 这个对 `approximate_size()` 函数的调用给名为 `a_kilobyte_is_1024_bytes` 的参数指定了值 `False`，然后给名为 `size` 的参数指定了值 `4000`。（看到了没？我告诉过你顺序没有关系。）
4. 这个调用会失败，因为你在命名参数后面紧跟了一个非命名（位置的）的参数，这个一定不会工作。从左到右的读取参数列表，一旦你有一个命名的参数，剩下的参数也必须是命名的。
5. 这个调用也会失败，和前面一个调用同样的原因。是不是很惊讶？别忘了，你给名为 `size` 的参数传入了值 `4000`，那么“显然的” `False` 这个值意味着对应了 `a_kilobyte_is_1024_bytes` 参数。但是 Python 不按照这种方式工作。只要你有一个命名参数，它右边的所有参数也都需要是命名参数。

编写易读的代码

我不会长期指手划脚的来烦你，解释给你的代码添加文档注释的重要性。只要知道代码被编写一次但是会被阅读很多次，而且你的代码最要的读者就是你自己，在编写它的六个月以后（例如，当你忘记了所有的东西但是又需要去修正一些东西的时候）。Python 使得编写易读的代码非常容易，因此要利用好这个优势。六个月以后你将会感谢我。

文档字符串

你可以通过使用一个文档字符串（简称 `docstring`）的方式给 Python 添加文档注释。在这个程序中，这个 `approximate_size()` 函数有一个 `docstring`：

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
```

每个函数都值得有一个合适的 docstring （文档字符串）。

三重引号表示一个多行的字符串。在开始引号和结束引号之间的所有东西都属于一个单独的字符串的一部分，包括回车、前导空格、和其他引号字符。你可以在任何地方使用它们，但是你会发现大部分时候它们在定义 `docstring` （文档注释）的时候使用。

> ☞三重引号也是一种容易的方法，用来定义一个同时包含单引号和双引号的字符串，就像 Perl 5 里面的 `qq/.../` 一样。

三重引号之间的所有东西都是这个函数的 `docstring` （文档字符串），用来用文档描述这个函数是做什么的。一个 `docstring` （文档字符串），如果有的话，必须是一个函数里面定义的第一个东西（也就是说，紧跟着函数声明的下一行）。 你不需要严格的给你的每个函数提供一个 `docstring` （文档字符串），但大部分时候你总是应该提供。我知道你在曾经使用过的每一种程序语言里面听说过这个，但是 Python 给你提供了额外的诱因：这个 `docstring` （文档字符串）就像这个函数的一个属性一样在运行时有效。

> ☞很多 Python 的集成开发环境（IDE）使用 `docstring` （文档字符串）来提供上下文敏感的文档，以便于当你输入一个函数名称的时候，它的 `docstring` 会以一个提示文本的方式显式出来。这可能会极其有用，但它只有在你写出好的 `docstring` （文档字符串）的时候才有用。

`import` 的搜索路径

在进一步讲解之前，我想简要的说一下库的搜索路径。当你试图导入（import）一个模块的时候，Python 会寻找几个地方。具体来说，它会搜寻在 `sys.path` 里面定义的所有目录。这只是一个列表，你可以容易地查看它或者使用标准的列表方法去修改它。（在内置数据类型你会了解更多关于列表的信息。）

```
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']

<module 'sys' (built-in)>

['/home/mark/diveintopython3/examples',
 '',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
```

1. 导入 `sys` 模块，使它的所有函数和属性可以被使用。
2. `sys.path` 是一个目录名称的列表，它构成了当前的搜索路径。（你会看到不一样的结果，这取决于你的操作系统，你正在运行的 Python 的版本，以及它原来被安装的位置。） Python 会从头到尾的浏览这些目录（按照这个顺序），寻找一个和你正要导入的模块名称匹配的 `.py` 文件。
3. 其实，我说谎了。真实情况比那个更加复杂，因为不是所有的模块都按照 `.py` 文件来存储。有些，比如 `sys` 模块，属于内置模块（*built-in modules*）， 他们事实上被置入到

Python 本身里面了。 内置模块使用起来和常规模块一样，但是无法取得它们的 Python 源代码，因为它们不是用 Python 写的！（ `sys` 模块是用 C 语言写的。）

4. 通过添加一个目录名称到 `sys.path` 里，你可以在运行时添加一个新的目录到 Python 的搜索路径中，然后无论任何时候你想导入一个模块，Python 都会同样的去查找那个目录。只要 Python 在运行，都会一直有效。
5. 通过使用 `sys.path.insert(0,` new_path `)`，你可以插入一个新的目录到 `sys.path` 列表的第一项，从而使其出现在 Python 搜索路径的开头。这几乎总是你想要的。万一出现名字冲突（例如，Python 自带了版本 2 的一个特定的库，但是你想使用版本 3），这个方法就能确保你的模块能够被发现和使用，替代 Python 自带的版本。

一切都是对象

假如你还不了解，我重复一下，我刚刚说过 Python 函数有属性，并且那些属性在运行时是可用的。一个函数，就像 Python 里面所有其他东西一样，是一个对象。

运行交互式的 Python Shell，按照下面的执行：

```
4.0 KiB

Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string
```

1. 第一行导入了作为一个模块的 `humansize` 程序 — 我们可以交互式的使用的一大块代码，或者来自于一个更大的 Python 程序。一旦你导入了一个模块，你就可以引用它的任何公有的函数、类、或者属性。模块可以通过这种方式访问其他模块的功能，同样的你也可以在 Python 交互式的 Shell 里面做这样的事情。这是一个重要的概念，贯穿这本书，你会看到更多的关于它的内容。
2. 当你想使用在导入的模块中定义的函数的时候，你需要包含模块的名称。因此你不能仅仅指明 `approximate_size` ，它必须是 `humansize.approximate_size` 才行。如果你曾经使用过 Java 里面的类，你就会依稀的感觉到这种方式比较熟悉。
3. 除了按照你期望的方式调用这个函数，你查看了这个函数的其中一个属性： `__doc__` 。

> ☞Python 里面的 `import` 就像 Perl 里面的 `require` 。一旦你导入（ `import` ）了一个 Python 模块，你就可以通过 `module`.`function` 的方式访问它的函数；一旦你要求（ `require` ）了一个 Perl 模块，你就可以通过 `module`::`function` 的方式访问它的函数。

什么是一个对象？

Python 里面的所有东西都是对象，所有东西都可以有属性和方法。所有函数都有一个内置的属性 `__doc__` ，用来返回这个函数的源代码里面定义的文档字符串（ `docstring` ）。 `sys` 模块是一个对象，它有（除了别的以外）一个名叫 `path` 的属性，等等。

不过，这还是没有回答这个更基础的问题：什么是一个对象？不同的程序语言用不同的方式定义了“对象”。在有些地方，它意味着所有的对象必须要有属性和方法；在另一些地方，它意味着所有的对象都是可衍生（可以创建子类）的。在 Python 里面，定义更加宽松。有些对象既没有属性也没有方法，然而它可以有。不是所有的对象都是可衍生的。但是，所有的东西都是对象，从这个意义上说，它能够被赋值到一个变量或者作为一个参数传入一个函数。

你可能从其他程序语言环境中听说过 “first-class object” 的说法。在 Python 中，函数是 *first-class objects*，你可以将一个函数作为一个参数传递给另外一个函数；模块是 *first-class objects*，你可以把整个模块作为一个参数传递给一个函数；类是 first-class objects，而且类的单独的实例也是 first-class objects。

这个很重要，因此刚开始我会重复几次以防你忘记了：在 *Python* 里面所有东西都是对象。字符串是对象，列表是对象，函数是对象，类是对象，类的实例是对象，甚至模块也是对象。

代码缩进

Python 函数没有明确的开始（ `begin` ）或者结束（ `end` ），也没有用大括号来标记函数从哪里开始从哪里停止。唯一的定界符是一个冒号（ `:` ）和代码自身的缩进。

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000

    size /= multiple
    if size < multiple:
        return '{0:.1f} {1}'.format(size, suffix)

raise ValueError('number too large')
```

1. 代码块是通过它们的缩进来定义的。我说的“代码块”，意思是指函数， `if` 语句、 `for` 循环、 `while` 循环，等等。 缩进表示一个代码块的开始，非缩进表示一个代码的结束。没有明确的大括号、中括号、或者关键字。这意味着空白很重要，而且必须要是一致的。在这个例子中，这个函数按照四个空格缩进。它不需要一定是四个空格，只是需要保持一致。第一个没有缩进的行标记了这个函数的结束。
2. 在 Python 中，一个 `if` 语句后面紧跟了一个代码块。如果 `if` 表达式的值为 true 则缩进的代码会被执行，否则它会跳到 `else` 代码块（如果有的话）。注意表达式的周围没有括号。
3. 这一行在 `if` 代码块里面。这个 `raise` 语句将抛出一个异常（类型是 `ValueError` ），但只有在 `size < 0` 的时候才抛出。
4. 这不是函数的结尾。完全空白的行不算。它们使代码更加易读，但它们不算作代码块的定界符。这个函数在下一行继续。

5. 这个 `for` 循环也标记了一个代码块的开始。代码块可以包含多行，只要它们都按照同样的数额缩进。这个 `for` 循环里面有三行。对于多行的代码块，也没有其他特殊的语法，只要缩进就可以了。

在刚开始的一些反对声和一些类比到 Fortran 的嘲笑之后，你将会平和的看待这个并开始领会到它的好处。一个主要的好处是所有的 Python 程序看起来都类似，因为缩进是一个语言的要求，不是一个风格的问题。这使得阅读和理解其他人的 Python 代码更加容易。

> ☞Python 使用回车符来分割语句，使用一个冒号和缩进来分割代码块。 C++ 和 Java 使用分号来分割语句，使用大括号来分割代码块。

异常

异常在 Python 中无处不在。事实上在标准 Python 库里面的每个模块都使用它们，而且在很多不同情形下， Python 自身也会抛出异常。贯穿这本书，你会反复的看到它们。

什么是一个异常？通常情况下，它是一个错误，提示某个东西出问题了。（不是所有的异常都是错误，但目前来说别担心那个） 某些程序语言鼓励对错误返回代码的使用，你可以对它进行检查。 Python 鼓励对异常的使用，你可以对它进行处理。

当一个错误发生在 Python Shell 里面的时候，它会打印一些关于这个异常以及它如何发生的详细信息，就此而已。这个被称之为一个 未被处理 的异常。在这个异常被抛出的时候，没有代码注意到并处理它，因此它把它的路径冒出来，返回到 Python Shell 的最顶层，输出一些调试信息，然后圆满结束。在这个 Shell 中，这没什么大不了的，但是如果在你的实际 Python 程序正在运行的时候发生，并且对这个异常没有做任何处理的话，整个程序就会嘎的一声停下来。可能那正是你想要的，也可能不是。

> ☞不像 Java， Python 函数不声明它们可能会抛出哪些异常。它取决于你去判断哪些可能的异常是你需要去捕获的。

一个异常不会造成整个程序崩溃。不过，异常是可以被处理的。有时候一个异常是真正地由于你代码里面的一个 bug 所引起的（比如访问一个不存在的变量），但有时候一个 异常是你可以预料到的东西。如果你在打开一个文件，它有可能不存在。如果你在导入一个模块，它可能没有被安装。如果你在连接到一个数据库，它有可能是无效的，或者你可能没有访问它需要的安全认证信息。如果你知道某行代码可能抛出一个异常，你应该使用 `try...except` 块来处理这个异常。

> ☞Python 使用 `try...except` 块来处理异常，使用 `raise` 语句来抛出异常。 Java 和 C++ 使用 `try...catch` 块来处理异常，使用 `throw` 语句来抛出异常。

这个 `approximate_size()` 函数在两个不同的情况下抛出异常：如果给定的 `size` 的值大于这个函数打算处理的值，或者如果它小于零。

```
if size < 0:
    raise ValueError('number must be non-negative')
```

抛出一个异常的语法足够简单。使用 `raise` 语句，紧跟着异常的名称，和一个人们可以读取的字符串用来调试。这个语法让人想起调用的函数。（实际上，异常是用类来实现的，这个 `raise` 语句事实上正在创建一个 `ValueError` 类的实例并传递一个字符串 `'number must be non-negative'` 到它的初始化方法里面。但是，我们已经有些超前了！）

> ☞你不需要在抛出异常的函数里面去处理它。如果一个函数没有处理它，这个异常会被传递到它的调用函数，然后那个函数的调用函数，等等"在这个堆栈上面。" 如果这个异常从来没有被处理，你的程序将会崩溃， Python 将会打印一个 "traceback" 的标准错误信息，并以此结束。这也可能正是你想要的，它取决于你的程序具体做什么。

捕获导入错误

其中一个 Python 的内置异常是 `ImportError`，它会在你试图导入一个模块并且失败的时候抛出。这有可能由于多种原因引起，但是最简单的情况是当在你的 import 搜索路径里面找不到这个模块的时候会发生。你可以用这个来包含可选的特性到你的程序中。例如，这个 `chardet` 库 提供字符编码自动检测。也许你的程序想在这个库存在的时候使用它，但是如果用户没有安装，也会优雅地继续执行。你可以使用 `try..except` 块来做这样的事情。

```
<mark>try</mark>:
  import chardet
<mark>except</mark> ImportError:
  chardet = None
```

然后，你可以用一个简单的 `if` 语句来检查 `chardet` 模块是否存在：

```
if chardet:
  # do something
else:
  # continue anyway
```

另一个对 `ImportError` 异常的通常使用是当两个模块实现了一个公共的 API，但我们更想要其中一个的时候。（可能它速度更快，或者使用了更少的内存。） 你可以试着导入其中一个模块，并且在这个模块导入失败的时候退回到另一个不同的模块。例如， XML 的章节谈论了两个模块实现一个公共的 API，叫做 `ElementTree` API。 第一个， `lxml` 是一个第三方的模块，你需要自己下载和安装。第二个， `xml.etree.ElementTree` 比较慢，但属于 Python 3 标准库的一部分。

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

在这个 `try..except` 块的结尾，你导入了某个模块并取名为 `etree` 。由于两个模块实现了一个公共的 API，你剩下的代码不需要一直去检查哪个模块被导入了。而且由于这个一定会被导入的模块总是叫做 `etree` ，你余下的代码就不会被调用不同名称模块的 `if` 语句所打乱。

Unbound 变量

再看看 `approximate_size()` 函数里面的这行代码：

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

你从不声明这个 `multiple` 变量，你只是给它赋值了。这样就可以了，因为 Python 让你那样做。 Python 将不会让你做的是，引用了一个变量，但从不给它赋值。这样的尝试将会抛出一个 `NameError` 的异常。

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 1
>>> x
1
```

将来有一天，你会因为这个而感谢 Python 。

所有的东西都是区分大小写的

Python 里面所有的名称都是区分大小写的：变量名、函数名、类名、模块名称、异常名称。如果你可以获取它、设置它、调用它、构建它、导入它、或者抛出它，那么它就是区分大小写的。

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'AN_INTEGER' is not defined
>>> An_Integer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'An_Integer' is not defined
>>> an_inteGer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'an_inteGer' is not defined
```

等等。

运行脚本

Python 里面所有东西都是对象。

Python 模块是对象，并且有几个有用的属性。在你编写它们的时候，通过包含一个特殊的仅在你从命令行运行 Python 文件的时候执行的代码块，你可以使用这些属性容易地测试你的模块。看看 `humansize.py` 的最后几行代码：

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

> ☞像 C 语言一样， Python 使用 `==` 来做比较，用 `=` 来赋值。不同于 C 语言的是，Python 不支持内嵌的赋值，所以没有机会出现你本以为在做比较而且意外的写成赋值的情况。

那么是什么使得这个 `if` 语句特别的呢？好吧，模块是对象，并且所有模块都有一个内置的属性 `__name__` 。一个模块的 `__name__` 属性取决于你怎么来使用这个模块。如果你 `import` 这个模块，那么 `__name__` 就是这个模块的文件名，不包含目录的路径或者文件的扩展名。

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

但是你也可以当作一个独立的程序直接运行这个模块，那样的话 `__name__` 将是一个特殊的默认值 `__main__` 。Python 将会评估这个 `if` 语句，寻找一个值为 true 的表达式，然后执行这个 `if` 代码块。在这个例子中，打印两个值。

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

这就是你的第一个 Python 程序！

深入阅读

- PEP 257: Docstring 约定解释了用什么来从大量的 `docstring` 中分辨出一个好的 `docstring` 。
- Python 教程：文档字符串也略微提到了这个主题。
- PEP 8: Python 代码的风格指南讨论了好的缩进风格。
- Python 参考手册解释了为什么说 Python 里面所有东西都是对象，因为有些人是书呆子，喜欢详细地讨论一些东西。

Chapter 2 内置数据类型

> " Wonder is the foundation of all philosophy, inquiry its progress, ignorance its end. " — Michel de Montaigne

深入

让我们暂时将 第一份 Python 程序 抛在脑后，来聊一聊数据类型。在 Python 中， 每个值都有一种数据类型，但您并不需要声明变量的数据类型。那该方式是如何运作的呢？Python 根据每个变量的初始赋值情况分析其类型，并在内部对其进行跟踪。

Python 有多种内置数据类型。以下是比较重要的一些：

1. **Booleans**［布尔型］ 或为 `True［真］` 或为 `False［假］` 。
2. **Numbers**［数值型］ 可以是 Integers［整数］ （ `1` 和 `2` ）、Floats［浮点数］ （ `1.1` 和 `1.2` ）、Fractions［分数］ （ `1/2` 和 `2/3` ）；甚至是 Complex Number［复数］ 。
3. **Strings**［字符串型］ 是 Unicode 字符序列，例如： 一份 HTML 文档。
4. **Bytes**［字节］ 和 **Byte Arrays**［字节数组］， 例如: 一份 JPEG 图像文件。
5. **Lists**［列表］ 是值的有序序列。
6. **Tuples**［元组］ 是有序而不可变的值序列。
7. **Sets**［集合］ 是装满无序值的包裹。
8. **Dictionaries**［字典］ 是键值对的无序包裹。

当然，还有更多的类型。在 Python 中一切均为对象，因此存在像 *module*［模块］、*function*［函数］、 *class*［类］、 *method*［方法］、 *file*［文件］ 甚至 *compiled code*［已编译代码］ 这样的类型。您已经见过这样一些例子：模块的 name、 函数的 `docstrings` 等等。将学到的包括 《类 与 迭代器》 中的 Classes［类］，以及 《文件》 中的 Files［文件］。

Strings［字符串］和 Bytes［字节串］比较重要，也相对复杂，足以开辟独立章节予以讲述。让我们先看看其它类型。

布尔类型

在布尔类型上下文中，您几乎可以使用任何表达式。

布尔类型或为真或为假。Python 有两个被巧妙地命名为 `True` 和 `False` 的常量，可用于对布尔类型的直接赋值。表达式也可以计算为布尔类型的值。在某些地方（如 `if` 语句），Python 所预期的就是一个可计算出布尔类型值的表达式。这些地方称为 布尔类型上下文环

境。事实上，可在布尔类型上下文环境中使用任何表达式，而 Python 将试图判断其真值。在布尔类型上下文环境中，不同的数据类型对于何值为真、何值为假有着不同的规则。（看过本章稍后的实例后，这一点将更好理解。）

例如，看看 `humansize.py` 中的这个片段：

```
if size < 0:
    raise ValueError('number must be non-negative')
```

`size` 是整数， 0 是整数，而 `<` 是数字运算符。 `size < 0` 表达式的结果始终是布尔值。可在 Python 交互式 shell 中自行测试下结果：

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

由于 Python 2 的一些遗留问题，布尔值可以当做数值对待。 `True` 为 `1` ； `False` 为 0 。

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

喔，喔，喔！别那么干。忘掉我刚才说的。

数值类型

数值类型是可畏的。有太多类型可选了。Python 同时支持 Integer［整型］ 和 Floating Point［浮点型］ 数值。无任何类型声明可用于区分；Python 通过是否有 小数 点来分辨它们。

```
<class 'int'>

True

2

2.0
>>> type(2.0)
<class 'float'>
```

1. 可以使用 `type()` 函数来检测任何值或变量的类型。正如所料，`1` 为 `int` 类型。
2. 同样，还可使用 `isinstance()` 函数判断某个值或变量是否为给定某个类型。
3. 将一个 `int` 与一个 `int` 相加将得到一个 `int` 。
4. 将一个 `int` 与一个 `float` 相加将得到一个 `float` 。Python 把 `int` 强制转换为 `float` 以进行加法运算；然后返回一个 `float` 类型的结果。

将整数强制转换为浮点数及反向转换

正如刚才所看到的，一些运算符（如：加法）会根据需把整数强制转换为浮点数。也可自行对其进行强制转换。

```
2.0

2

2

-2

1.1234567890123457

<class 'int'>
```

1. 通过调用 `float()` 函数，可以显示地将 `int` 强制转换为 `float` 。
2. 毫不出奇，也可以通过调用 `int()` 将 `float` 强制转换为 `int` 。
3. `int()` 将进行取整，而不是四舍五入。
4. 对于负数，`int()` 函数朝着 0 的方法进行取整。它是个真正的取整（截断）函数，而不是 floor［地板］函数。
5. 浮点数精确到小数点后 15 位。
6. 整数可以任意大。

> ☞Python 2 对于 `int [整型]` 和 `long [长整型]` 采用不同的数据类型。`int` 数据类型受到 `sys.maxint` 的限制，因平台该限制也会有所不同，但通常是 `2**32-1` 。Python 3 只有一种整数类型，其行为方式很有点像 Python 2 的旧 `long [长整数]` 类型。参阅 PEP 237 了解更多细节。

常见数值运算

对数值可进行各种类型的运算。

```
5.5

5

-6

5.0

121

1
```

1. `/` 运算符执行浮点除法。即便分子和分母都是 `int`，它也返回一个 `float` 浮点数。
2. `//` 运算符执行古怪的整数除法。如果结果为正数，可将其视为朝向小数位取整（不是四舍五入），但是要小心这一点。
3. 当整数除以负数， `//` 运算符将结果朝着最近的整数“向上”四舍五入。从数学角度来说，由于 `-6` 比 `-5` 要小，它是“向下”四舍五入，如果期望将结果取整为 `-5`，它将会误导你。
4. `//` 运算符并非总是返回整数结果。如果分子或者分母是 `float`，它仍将朝着最近的整数进行四舍五入，但实际返回的值将会是 `float` 类型。
5. `**` 运算符的意思是“计算幂”， `11**2` 结果为 `121` 。
6. `%` 运算符给出了进行整除之后的余数。 `11` 除以 `2` 结果为 `5` 以及余数 `1`，因此此处的结果为 `1` 。

> ☞在 Python 2 中，运算符 `/` 通常表示整数除法，但是可以通过在代码中加入特殊指令，使其看起来像浮点除法。在 Python 3 中， `/` 运算符总是表示浮点除法。参阅 PEP 238 了解更多细节。

分数

Python 并不仅仅局限于整数和浮点数类型。它可以完成你在高中阶段学过、但几乎已经全部忘光的所有古怪数学运算。

```
>>> x
Fraction(1, 3)

Fraction(2, 3)

Fraction(3, 2)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fractions.py", line 96, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
```

1. 为启用 fractions 模块，必先引入 `fractions` 模块。

2. 为定义一个分数，创建一个 `Fraction` 对象并传入分子和分母。
3. 可对分数进行所有的常规数学计算。运算返回一个新的 `Fraction` 对象。`2 * (1/3) = (2/3)`
4. `Fraction` 对象将会自动进行约分。`(6/4) = (3/2)`
5. 在杜绝创建以零为分母的分数方面，Python 有着良好的敏感性。

三角函数

还可在 Python 中进行基本的三角函数运算。

```
>>> import math

3.1415926535897931

1.0

0.99999999999999989
```

1. `math` 模块中有一个代表 π 的常量，表示圆的周长与直径之比率（圆周率）。
2. `math` 模块包括了所有的基本三角函数，包括：`sin()` 、 `cos()` 、 `tan()` 及像 `asin()` 这样的变体函数。
3. 然而要注意的是 Python 并不支持无限精度。`tan(π / 4)` 将返回 `1.0` ，而不是 `0.99999999999999989` 。

布尔上下文环境中的数值

零值是 false［假］，非零值是 true［真］。

可以在 `if` 这样的 布尔类型上下文环境中 使用数值。零值是 false［假］，非零值是 true［真］。

```

...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...

yes, it's true
>>> is_it_true(-1)
yes, it's true
>>> is_it_true(0)
no, it's false

yes, it's true
>>> is_it_true(0.0)
no, it's false
>>> import fractions

yes, it's true
>>> is_it_true(fractions.Fraction(0, 1))
no, it's false
```

1. 您知道可以在 Python 交互式 Shell 中定义自己的函数吗？只需在每行的结尾按 `回车键` ，然后在某一空行按 `回车键` 结束。
2. 在布尔类型上下文环境中，非零整数为真；零为假。
3. 非零浮点数为真； `0.0` 为假。请千万小心这一点！如果有轻微的四舍五入偏差（正如在前面小节中看到的那样，这并非不可能的事情），那么 Python 将测试 `0.0000000000001` 而不是 0 ，并将返回一个 `True` 值。
4. 分数也可在布尔类型上下文环境中使用。无论 `n` 为何值， `Fraction(0, n)` 为假。所有其它分数为真。

列表

列表是 Python 的主力数据类型。当提到 “列表 ”时，您脑海中可能会闪现“必须进一步声明大小的数组，只能包含同一类对象“ 等想法。千万别这么想。列表比那要酷得多。

> ☞ Python 中的列表类似 Perl 5 中的数组。在 Perl 5 中，存储数组的变量总是以字符 `@` 开头；在 Python 中， 变量可随意命名，Python 仅在内部对数据类型进行跟踪。
>
> ☞ Python 中的列表更像 Java 中的数组（尽管可以把列表当做生命中所需要的一切来使用)。一个更好的比喻可能是 `ArrayList` 类，该类可以容纳任何对象，并可在添加新元素时进行动态拓展。

创建列表

列表创建非常轻松：使用中括号包裹一系列以逗号分割的值即可。

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']

'a'

'example'

'example'

'mpilgrim'
```

1. 首先，创建一个包含 5 个元素的列表。要注意的是它们保持了最初的顺序。这并不是偶然的。列表是元素的有序集合。
2. 列表可当做以零为基点的数组使用。非空列表的首个元素始终是 `a_list[0]` 。
3. 该 5 元素列表的最后一个元素是 `a_list[4]`，因为列表（索引）总是以零为基点的。
4. 使用负索引值可从列表的尾部向前计数访问元素。任何非空列表的最后一个元素总是 `a_list[-1]` 。
5. 如果负数令你混淆，可将其视为如下方式： `a_list[- n ] == a_list[len(a_list) - n ]` 。因此在此列表中， `a_list[-3] == a_list[5 - 3] == a_list[2]` 。

列表切片

a_list[0] 是列表的第一个元素。

定义列表后，可从其中获取任何部分作为新列表。该技术称为对列表进行 切片 。

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']

['b', 'mpilgrim']

['b', 'mpilgrim', 'z']

['a', 'b', 'mpilgrim']

['a', 'b', 'mpilgrim']

['z', 'example']

['a', 'b', 'mpilgrim', 'z', 'example']
```

1. 通过指定两个索引值，可以从列表中获取称作“切片”的某个部分。返回值是一个新列表，它包含列表(??切片)中所有元素，按顺序从第一个切片索引开始（本例中为 `a_list[1]` ），截止但不包含第二个切片索引（本例中的 `a_list[3]` ）。
2. 如果切片索引之一或两者均为负数，切片操作仍可进行。如果有帮助的话，您可以这么思考：自左向右读取列表，第一个切片索引指明了想要的第一个元素，第二个切片索引指明了第一个不想要的元素。返回值是两者之间的任何值。 between.
3. 列表是以零为起点的，因此 `a_list[0:3]` 返回列表的头三个元素，从 `a_list[0]` 开始，截止到但不包括 `a_list[3]` 。
4. 如果左切片索引为零，可以将其留空而将零隐去。因此 `a_list[:3]` 与 `a_list[0:3]` 是完全相同的，因为起点 0 被隐去了。
5. 同样，如果右切片索引为列表的长度，也可以将其留空。因此 `a_list[3:]` 与 `a_list[3:5]` 是完全相同的，因为该列表有五个元素。此处有个好玩的对称现象。在这个五元素列表中， `a_list[:3]` 返回头三个元素，而 `a_list[3:]` 返回最后两个元素。事实上，无论列表的长度是多少，`a_list[:` n `]` 将返回头 `n` 个元素，而 `a_list[` n `:]` 返回其余部分。
6. 如果两个切片索引都留空，那么将包括列表所有的元素。但该返回值与最初的 `a_list` 变量并不一样。它是一个新列表，只不过恰好拥有完全相同的元素而已。 `a_list[:]` 是对列表进行复制的一条捷径。

向列表中新增项

有四种方法可用于向列表中增加元素。

```
>>> a_list = ['a']

['a', 2.0, 3]

>>> a_list
['a', 2.0, 3, True]

>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']

>>> a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

1. `+` 运算符连接列表以创建一个新列表。列表可包含任何数量的元素；没有大小限制（除了可用内存的限制）。然而，如果内存是个问题，那就必须知道在进行连接操作时，将在内存中创建第二个列表。在该情况下，新列表将会立即被赋值给已有变量 `a_list` 。因此，实际上该行代码包含两个步骤 — 连接然后赋值 — 当处理大型列表时，该操作可能（暂时）消耗大量内存。
2. 列表可包含任何数据类型的元素，单个列表中的元素无须全为同一类型。下面的列表中包含一个字符串、一个浮点数和一个整数。
3. `append()` 方法向列表的尾部添加一个新的元素。（现在列表中有 四种 不同数据类型！）
4. 列表是以类的形式实现的。“创建”列表实际上是将一个类实例化。因此，列表有多种方法可以操作。 `extend()` 方法只接受一个列表作为参数，并将该参数的每个元素都添加到原有的列表中。
5. `insert()` 方法将单个元素插入到列表中。第一个参数是列表中将被顶离原位的第一个元素的位置索引。列表中的元素并不一定要是唯一的；比如说：现有两个各自独立的元素，其值均为 `'Ω'` :，第一个元素 `a_list[0]` 以及最后一个元素 `a_list[6]` 。

> ☞ `a_list`.insert(0, `value`) 就像是 `Perl` 中的unshift()` 函数。它将一个元素添加到列表的头部，所有其它的元素都被顶理原先的位置以腾出空间。

让我们进一步看看 `append()` 和 `extend()` 的区别。

```
>>> a_list = ['a', 'b', 'c']

>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']

6
>>> a_list[-1]
'f'

>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]

7
>>> a_list[-1]
['g', 'h', 'i']
```

1. `extend()` 方法只接受一个参数，而该参数总是一个列表，并将列表 `a_list` 中所有的元素都添加到该列表中。

2. 如果开始有个 3 元素列表，然后将它与另一个 3 元素列表进行 extend 操作，结果是将获得一个 6 元素列表。
3. 另一方面， `append()` 方法只接受一个参数，但可以是任何数据类型。在此，对一个 3 元素列表调用 `append()` 方法。
4. 如果开始的时候有个 6 元素列表，然后将一个列表 append［添加］上去，结果就会……得到一个 7 元素列表。为什么是 7 个？因为最后一个元素（刚刚 append［添加］ 的元素） 本身是个列表 。列表可包含任何类型的数据，包括其它列表。这可能是你所需要的结果，也许不是。但如果这就是你想要的，那这就是你所得到的。

在列表中检索值

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']

2

True
>>> 'c' in a_list
False

3

2

Traceback (innermost last):
  File "<interactive input>", line 1, in ?ValueError: list.index(x): x not in list
```

1. 如你所期望， `count()` 方法返回了列表中某个特定值出现的次数。
2. 如果你想知道的是某个值是否出现在列表中， `in` 运算符将会比使用 `count()` 方法要略快一些。 `in` 运算符总是返回 `True` 或 `False` ；它不会告诉你该值出现在什么位置。
3. 如果想知道某个值在列表中的精确位置，可调用 `index()` 方法。尽管可以通过第二个参数（以 0 为基点的）索引值来指定起点，通过第三个参数（以 0 基点的）索引来指定搜索终点，但缺省情况下它将搜索整个列表，
4. `index()` 方法将查找某值在列表中的第一次出现。在该情况下， `'new'` 在列表中出现了两次，分别为 `a_list[2]` 和 `a_list[4]` ，但 `index()` 方法将只返回第一次出现的位置索引值。
5. 可能 出乎 您的预期，如果在列表中没有找到该值， `index()` 方法将会引发一个例外。

等等，什么？是这样的：如果没有在列表中找到该值， `index()` 方法将会引发一个例外。这是 Python 语言最显著不同之处，其它多数语言将会返回一些无效的索引值（像是 `-1` ）。当然，一开始这一点看起来比较讨厌，但我想您会逐渐欣赏它。这意味着您的程序将会在问题的源头处崩溃，而不是之后奇怪地、默默地崩溃。请记住， `-1` 是合法的列表索引值。如果 `index()` 方法返回 `-1` ，可能会导致调整过程变得不那么有趣！

从列表中删除元素

列表永远不会有缝隙。

列表可以自动拓展或者收缩。您已经看到了拓展部分。也有几种方法可从列表中删除元素。

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list[1]
'b'

>>> a_list
['a', 'new', 'mpilgrim', 'new']

'new'
```

1. 可使用 `del` 语句从列表中删除某个特定元素。
2. 删除索引 `1` 之后再访问索引 `1` 将 不会 导致错误。被删除元素之后的所有元素将移动它们的位置以“填补”被删除元素所产生的“缝隙”。

不知道位置索引？这不成问题，您可以通过值而不是索引删除元素。

```

>>> a_list
['a', 'mpilgrim', 'new']

>>> a_list
['a', 'mpilgrim']
>>> a_list.remove('new')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

1. 还可以通过 `remove()` 方法从列表中删除某个元素。 `remove()` 方法接受一个 *value* 参数，并删除列表中该值的第一次出现。同样，被删除元素之后的所有元素将会将索引位置下移，以“填补缝隙”。列表永远不会有“缝隙”。
2. 您可以尽情地调用 `remove()` 方法，但如果试图删除列表中不存在的元素，它将引发一个例外。

Removing Items From A List: Bonus Round

另一有趣的列表方法是 `pop()` 。 `pop()` 方法是从列表删除元素的另一方法，但有点变化。

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim']

'mpilgrim'
>>> a_list
['a', 'b', 'new']

'b'
>>> a_list
['a', 'new']
>>> a_list.pop()
'new'
>>> a_list.pop()
'a'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

1. 如果不带参数调用， `pop()` 列表方法将删除列表中最后的元素，并返回所删除的值。
2. 可以从列表中 pop［弹出］任何元素。只需传给 `pop()` 方法一个位置索引值。它将删除该元素，将其后所有元素移位以“填补缝隙”,然后返回它删除的值。
3. 对空列表调用 `pop()` 将会引发一个例外。

> ☞不带参数调用的 `pop()` 列表方法就像 Perl 中的 `pop()` 函数。它从列表中删除最后一个元素并返回所删除元素的值。Perl 还有另一个函数 `shift()` ，可用于删除第一个元素并返回其值；在 Python 中，该函数相当于 `` `a_list `` .pop(0)` 。

布尔上下文环境中的列表

空列表为假；其它所有列表为真。

可以在 `if` 这样的 布尔类型上下文环境中 使用列表。

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...

no, it's false

yes, it's true

yes, it's true
```

1. 在布尔类型上下文环境中，空列表为假值。
2. 任何至少包含一个上元素的列表为真值。
3. 任何至少包含一个上元素的列表为真值。元素的值无关紧要。

元组

元素 是不可变的列表。一旦创建之后，用任何方法都不可以修改元素。

```
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')

'a'

'example'

('b', 'mpilgrim')
```

1. 元组的定义方式和列表相同，除了整个元素的集合都用圆括号，而不是方括号闭合。
2. 和列表一样，元组的元素都有确定的顺序。元组的索引也是以零为基点的，和列表一样，因此非空元组的第一个元素总是 `a_tuple[0]` 。

3. 负的索引从元组的尾部开始计数，这和列表也是一样的。
4. 和列表一样，元组也可以进行切片操作。对列表切片可以得到新的列表；对元组切片可以得到新的元组。

元组和列表的主要区别是元组不能进行修改。用技术术语来说，元组是 不可变更 的。从实践的角度来说，没有可用于修改元组的方法。列表有像 `append()`、 `extend()`、 `insert()`、 `remove()` 和 `pop()` 这样的方法。这些方法，元组都没有。可以对元组进行切片操作（因为该方法创建一个新的元组），可以检查元组是否包含了特定的值（因为该操作不修改元组），还可以……就那么多了。

```
# continued from the previous example
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')

Traceback (innermost last):
  File "<interactive input>", line 1, in ?AttributeError: 'tuple' object has no attribute

Traceback (innermost last):
  File "<interactive input>", line 1, in ?AttributeError: 'tuple' object has no attribute

4

True
```

1. 无法向元组添加元素。元组没有 `append()` 或 `extend()` 方法。
2. 不能从元组中删除元素。元组没有 `remove()` 或 `pop()` 方法。
3. 可以 在元组中查找元素，由于该操作不改变元组。
4. 还可以使用 `in` 运算符检查某元素是否存在于元组中。

那么元组有什么好处呢？

- 元组的速度比列表更快。如果定义了一系列常量值，而所需做的仅是对它进行遍历，那么请使用元组替代列表。
- 对不需要改变的数据进行“写保护”将使得代码更加安全。使用元组替代列表就像是有一条隐含的 `assert` 语句显示该数据是常量，特别的想法（及特别的功能）必须重写。（？？）
- 一些元组可用作字典键（特别是包含字符串、数值和其它元组这样的不可变数据的元组）。列表永远不能当做字典键使用，因为列表不是不可变的。

> ☞元组可转换成列表，反之亦然。内建的 `tuple()` 函数接受一个列表参数，并返回一个包含同样元素的元组，而 `list()` 函数接受一个元组参数并返回一个列表。从效果上看， `tuple()` 冻结列表，而 `list()` 融化元组。

布尔上下文环境中的元组

可以在 `if` 这样的 布尔类型上下文环境中 使用元组。

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...

no, it's false

yes, it's true

yes, it's true

<class 'bool'>
>>> type((False,))
<class 'tuple'>
```

1. 在布尔类型上下文环境中，空元组为假值。
2. 任何至少包含一个上元素的元组为真值。
3. 任何至少包含一个上元素的元组为真值。元素的值无关紧要。不过此处的逗号起什么作用呢？
4. 为创建单元素元组，需要在值之后加上一个逗号。没有逗号，Python 会假定这只是一对额外的圆括号，虽然没有害处，但并不创建元组。

同时赋多个值

以下是一种很酷的编程捷径：在 Python 中，可使用元组来一次赋多值。

```
>>> v = ('a', 2, True)

>>> x
'a'
>>> y
2
>>> z
True
```

1. `v` 是一个三元素的元组，而 `(x, y, z)` 是包含三个变量的元组。将其中一个赋值给另一个将会把 `v` 中的每个值按顺序赋值给每一个变量。

该特性有多种用途。假设需要将某个名称指定某个特定范围的值。可以使用内建的 `range()` 函数进行多变量赋值以快速地进行连续变量赋值。

```

0
>>> TUESDAY
1
>>> SUNDAY
6
```

1. 内建的 `range()` 函数构造了一个整数序列。（从技术上来说， `range()` 函数返回的既不是列表也不是元组，而是一个 迭代器，但稍后您将学到它们的区别。） `MONDAY` 、

TUESDAY 、 WEDNESDAY 、 THURSDAY 、 FRIDAY 、 SATURDAY 和 SUNDAY 是您所定义的变量。（本例来自于 calendar 模块，该短小而有趣的模块打印日历，有点像 UNIX 程序 cal 。该 calendar 模块为星期数定义了整数常量。
2. 现在，每个变量都有其值了： MONDAY 为 0， TUESDAY 为 1 ，如此类推。

还可以使用多变量赋值创建返回多值的函数，只需返回一个包含所有值的元组。调用者可将返回值视为一个简单的元组，或将其赋值给不同的变量。许多标准 Python 类库这么干，包括在下一章将学到的 os 模块。

集合

集合set 是装有独特值的无序“袋子”。一个简单的集合可以包含任何数据类型的值。如果有两个集合，则可以执行像联合、交集以及集合求差等标准集合运算。

创建集合

重中之重。创建集合非常简单。

```
>>> a_set
{1}

<class 'set'>

>>> a_set
{1, 2}
```

1. 要创建只包含一个值的集合，仅需将该值放置于花括号之间。（ {} ）。
2. 实际上，集合以 类 的形式实现，但目前还无须考虑这一点。
3. 要创建多值集合，请将值用逗号分开，并用花括号将所有值包裹起来。

还可以 列表 为基础创建集合。

```
>>> a_list = ['a', 'b', 'mpilgrim', True, False, 42]

{'a', False, 'b', True, 'mpilgrim', 42}

['a', 'b', 'mpilgrim', True, False, 42]
```

1. 要从列表创建集合，可使用 set() 函数。（懂得如何实现集合的学究可能指出这实际上并不是调用某个函数，而是对某个类进行实例化。我保证在本书稍后的地方将会学到其中的区别。目前而言，仅需知道 set() 行为与函数类似，以及它返回一个集合。）
2. 正如我之前提到的， 简单的集合可以包括任何数据类型的值。而且，如我之前所提到的，集合是 无序的。该集合并不记得用于创建它的列表中元素的最初顺序。如果向集合中添加元素，它也不会记得添加的顺序。
3. 初始的列表并不会发生变化。

还没有任何值？没有问题。可以创建一个空的集合。

```
set()

<class 'set'>

0

>>> type(not_sure)
<class 'dict'>
```

1. 要创建空集合，可不带参数调用 `set()` 。
2. 打印出来的空集合表现形式看起来有点儿怪。也许，您期望看到一个 `{}` 吧 ？该符号表示一个空的字典，而不是一个空的集合。本章稍后您将学到关于字典的内容。
3. 尽管打印出的形式奇怪，这 确实是 一个集合……
4. …… 同时该集合没有任何成员。
5. 由于从 Python 2 沿袭而来历史的古怪规定，不能使用两个花括号来创建空集合。该操作实际创建一个空字典，而不是一个空集合。

修改集合

有两种方法可向现有集合中添加值： `add()` 方法和 `update()` 方法。

```
>>> a_set = {1, 2}

>>> a_set
{1, 2, 4}

3

>>> a_set
{1, 2, 4}

3
```

1. `add()` 方法接受单个可以是任何数据类型的参数，并将该值添加到集合之中。
2. 该集合现在有三个成员了。
3. 集合是装 唯一值 的袋子。如果试图添加一个集合中已有的值，将不会发生任何事情。将不会引发一个错误；只是一条空操作。
4. 该集合 仍然 只有三个成员。

```
>>> a_set = {1, 2, 3}
>>> a_set
{1, 2, 3}

{1, 2, 3, 4, 6}

>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 13}

>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```

1. `update()` 方法仅接受一个集合作为参数，并将其所有成员添加到初始列表中。其行为方式就像是对参数集合中的每个成员调用 `add()` 方法。
2. 由于集合不能包含重复的值，因此重复的值将会被忽略。
3. 实际上，可以带任何数量的参数调用 `update()` 方法。如果调用时传递了两个集合，`update()` 将会被每个集合中的每个成员添加到初始的集合当中（丢弃重复值）。
4. `update()` 方法还可接受一些其它数据类型的对象作为参数，包括列表。如果调用时传入列表，`update()` 将会把列表中所有的元素添加到初始集合中。

从集合中删除元素

有三种方法可以用来从集合中删除某个值。前两种，`discard()` 和 `remove()` 有细微的差异。

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}

>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}

>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}

>>> a_set
{1, 3, 36, 6, 45, 15, 28}

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21
```

1. `discard()` 接受一个单值作为参数，并从集合中删除该值。
2. 如果针对一个集合中不存在的值调用 `discard()` 方法，它不进行任何操作。不产生错误；只是一条空指令。
3. `remove()` 方法也接受一个单值作为参数，也从集合中将其删除。
4. 区别在这里：如果该值不在集合中，`remove()` 方法引发一个 `KeyError` 例外。

就像列表，集合也有个 `pop()` 方法。

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}

1
>>> a_set.pop()
3
>>> a_set.pop()
36
>>> a_set
{6, 10, 45, 15, 21, 28}

>>> a_set
set()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

1. `pop()` 方法从集合中删除某个值，并返回该值。然而，由于集合是无序的，并没有“最后一个”值的概念，因此无法控制删除的是哪一个值。它基本上是随机的。
2. `clear()` 方法删除集合中 所有 的值，留下一个空集合。它等价于 `a_set = set()`，该语句创建一个新的空集合，并用之覆盖 `a_set` 变量的之前的值。
3. 试图从空集合中弹出某值将会引发 `KeyError` 例外。

常见集合操作

Python 的 `集合` 类型支持几种常见的运算。

```
>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}

True
>>> 31 in a_set
False
>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}

{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}

{9, 2, 12, 5, 21}

{195, 4, 76, 51, 30, 127}

{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

1. 要检测某值是否是集合的成员，可使用 `in` 运算符。其工作原理和列表的一样。
2. `union()` 方法返回一个新集合，其中装着 在两个 集合中出现的元素。
3. `intersection()` 方法返回一个新集合，其中装着 同时 在两个集合中出现的所有元素。
4. `difference()` 方法返回的新集合中，装着所有在 `a_set` 出现但未在 `b_set` 中的元素。
5. `symmetric_difference()` 方法返回一个新集合，其中装着所有 只在其中一个 集合中出现的元素。

这三种方法是对称的。

```
# continued from the previous example

{3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}

True

True

True

False
```

1. `a_set` 与 `b_set` 的对称差分 看起来 和 `b_set` 与 `a_set` 的对称差分不同，但请记住：集合是无序的。任何两个包含所有同样值（无一遗漏）的集合可认为是相等的。
2. 而这正是这里发生的事情。不要被 Python Shell 对这些集合的输出形式所愚弄了。它们包含相同的值，因此是相等的。

3. 对两个集合的 Union［并集］操作也是对称的。
4. 对两个集合的 Intersection［交集］操作也是对称的。
5. 对两个集合的 Difference［求差］操作不是对称的。这是有意义的；它类似于从一个数中减去另一个数。操作数的顺序会导致结果不同。

最后，有几个您可能会问到的问题。

```
>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}

True

True

>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False
```

1. `a_set` 是 `b_set` 的 子集 — 所有 `a_set` 的成员均为 `b_set` 的成员。
2. 同样的问题反过来说， `b_set` 是 `a_set` 的 超集，因为 `a_set` 的所有成员均为 `b_set` 的成员。
3. 一旦向 `a_set` 添加一个未在 `b_set` 中出现的值，两项测试均返回 `False` 。

布尔上下文环境中的集合

可在 `if` 这样的 布尔类型上下文环境中 使用集合。

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...

no, it's false

yes, it's true

yes, it's true
```

1. 在布尔类型上下文环境中，空集合为假值。
2. 任何至少包含一个上元素的集合为真值。
3. 任何至少包含一个上元素的集合为真值。元素的值无关紧要。

字典

字典 是键值对的无序集合。向字典添加一个键的同时，必须为该键增添一个值。（之后可随时修改该值。） Python 的字典为通过键获取值进行了优化，而不是反过来。

> ☞Python 中的字典与 Perl 5 中的 hash [散列]类似。在 Perl 5 中，散列存储的变量总是以一个 `%` 符开头。在 Python 中，变量可以随意命名，而 Python 内部跟踪其数据类型。

创建字典

创建字典非常简单。其语法与 集合 的类似，但应当指定键值对而不是值。有了字典后，可以通过键来查找值。

```
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}

'db.diveintopython3.org'

'mysql'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'db.diveintopython3.org'
```

1. 首先，通过将两个字典项指定给 `a_dict` 变量创建了一个新字典。每个字典项都是一组键值对，整个字典项集合都被大括号包裹在内。
2. `'server'` 为键，通过 `a_dict['server']` 引用的关联值为 `'db.diveintopython3.org'` 。
3. `'database'` 为键，通过 `a_dict['database']` 引用的关联值为 `'mysql'` 。
4. 可以通过键获取值，但不能通过值获取键。因此 `a_dict['server']` 为 `'db.diveintopython3.org'` ，而 `a_dict['db.diveintopython3.org']` 会引发例外，因为 `'db.diveintopython3.org'` 并不是键。

修改字典

字典没有预定义的大小限制。可以随时向字典中添加新的键值对，或者修改现有键所关联的值。继续前面的例子：

```
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}

>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'blog'}

{'server': 'db.diveintopython3.org', 'user': 'mark', 'database': 'blog'}

>>> a_dict
{'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}

>>> a_dict
{'User': 'mark', 'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
```

1. 在字典中不允许有重复的键。对现有的键赋值将会覆盖旧值。
2. 可随时添加新的键值对。该语法与修改现有值相同。

3. 新字典项（键为 `'user'`，值为 `'mark'`）出现在中间。事实上，在第一个例子中字典项按顺序出现是个巧合；现在它们不按顺序出现同样也是个巧合。
4. 对既有字典键进行赋值只会用新值替代旧值。
5. 该操作会将 `user` 键的值改回 "mark" 吗？不会！仔细看看该键——有个大写的 `U` 出现在 `"User"` 中。字典键是区分大小写的，因此该语句创建了一组新的键值对，而不是覆盖既有的字典项。对你来说它们可能是一样的，但对于 Python 而言它们是完全不同的。

混合值字典

字典并非只能用于字符串。字典的值可以是任何数据类型，包括整数、布尔值、任何对象，甚至是其它的字典。而且就算在同一字典中，所有的值也无须是同一类型，您可根据需要混合匹配。字典的键要严格得多，可以是字符串、整数和其它一些类型。在同一字典中也可混合、匹配使用不同数据类型的键。

实际上，您已经在 your first Python program 见过一个将非字符串用作键的字典了。

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

让我们在交互式 shell 中剖析一下：

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

2

True

['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']

['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']

'TB'
```

1. 类似 列表 和 集合 ，`len()` 函数将返回字典中键的数量。
2. 而且像列表和集合一样，可使用 `in` 运算符以测试某个特定的键是否在字典中。
3. `1000` 是 字典 `SUFFIXES` 的一个键；其值为一个 8 元素列表（确切地说，是 8 个字符串）。
4. 同样，`1024` 是字典 `SUFFIXES` 的键；其值也是一个 8 元素列表。
5. 由于 `SUFFIXES[1000]` 是列表，可以通过它们的 0 基点索引来获取列表中的单个元素。

布尔上下文环境中的字典

空字典为假值；所有其它字典为真值。

可以在 `if` 这样的 布尔类型上下文环境中 使用字典。

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...

no, it's false

yes, it's true
```

1. 在布尔类型上下文环境中，空字典为假值。
2. 至少包含一个键值对的字典为真值。

None

`None` 是 Python 的一个特殊常量。它是一个 空 值。 `None` 与 `False` 不同。 `None` 不是 0 。 `None` 不是空字符串。将 `None` 与任何非 `None` 的东西进行比较将总是返回 `False` 。

`None` 是唯一的空值。它有着自己的数据类型（ `NoneType` ）。可将 `None` 赋值给任何变量，但不能创建其它 `NoneType` 对象。所有值为 `None` 变量是相等的。

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True
```

布尔上下文环境中的 None

在 布尔类型上下文环境中， `None` 为假值，而 `not None` 为真值。

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(None)
no, it's false
>>> is_it_true(not None)
yes, it's true
```

深入阅读

- 布尔运算
- 数值类型
- 序列类型
- 集合类型
- 映射类型
- `fractions [分数]` 模块
- `math [数学]` 模块
- PEP 237: 统一长整数和整数
- PEP 238: 修改除法运算符

您在这里: 主页 ▸ 深入Python 3 ▸

Chapter 3 解析

> " Our imagination is stretched to the utmost, not, as in fiction, to imagine things which are not really there, but just to comprehend those things which are. " — Richard Feynman

深入

这一章节将围绕一个非常强大的技术向你介绍列表解析，字典解析和集合解析这三个概念。但是，我要先打个岔介绍两个帮助你浏览本地文件系统的模块。

处理文件和目录

Python 3 带有一个模块叫做 `os`，代表 "操作系统(operating system)。" `os` 模块 包含非常多的函数用于获取(和修改)本地目录、文件进程、环境变量等的信息。Python 尽最大的努力在所有支持的操作系统上提供一个统一的API， 这样你就可以在保证程序能够在任何的计算机上运行的同时尽量少的包含平台特定的代码。

当前工作目录

当你刚刚开始学习Python的时候， 你将花大量的时间在 Python Shell上。 在整本书中，你将一直看见类似下面的例子:

1. 在 `examples` 目录导入某一个模块
2. 调用模块的某一个函数
3. 解释输出结果

总是有一个当前工作目录

如果你不知道当前工作目录， 第一步很可能会得到一个 `ImportError` 。 为什么? 因为 Python 将在导入搜索路径中查找示例模块， 但是由于 `examples` 目录没有包含在搜索路径中，查找将失败。 你可以通过下面两个方法之一来解决这个问题:

1. 将 `examples` 目录加入到导入搜索路径中
2. 将当前工作目录切换到 `examples` 目录

Python在任何时候都在暗地里记住了当前工作目录这个属性。无论你是在Python Shell 中，还是在命令行运行你自己的Python 脚本，抑或是在Web 服务器上运行Python CGI 脚本，当前工作目录总是存在。

`os` 模块提供了两个函数处理当前工作目录

```
C:\Python31
C:\Users\pilgrim\diveintopython3\examples
```

1. `os` 是Python 自带的; 你可以在任何时间，任何地方导入它。
2. 使用 `os.getcwd()` 函数获得当前工作目录。当你运行一个图形化的Python Shell 时，当前工作目录默认将是Python Shell的可执行文件所在的目录。在Windows 上， 这个目录取决于你将Python安装在哪里; 默认位置是 `c:\Python31` 。如果你通过命令行运行Python Shell，当前工作目录是你运行 `python3` 时所在的目录。
3. 使用 `os.chdir()` 函数改变当前工作目录
4. 运行 `os.chdir()` 函数时，即使在Windows上，我也总是使用Linux风格的路径(正斜杠，没有盘符)。这就是Python 尝试隐藏操作系统差异的一个地方。

处理文件名和目录名

既然我们说到了目录，我得指出 `os.path` 模块。 `os.path` 模块包含了操作文件名和目录名的函数.

```
>>> import os
/Users/pilgrim/diveintopython3/examples/humansize.py
/Users/pilgrim/diveintopython3/examples\humansize.py
c:\Users\pilgrim
c:\Users\pilgrim\diveintopython3\examples\humansize.py
```

1. `os.path.join()` 函数从一个或多个路径片段中构造一个路径名。 在这个例子中， 它仅仅是简单的拼接字符串.
2. 这个例子稍微复杂一点， 在和文件名拼接前， `join` 函数给路径名添加一个额外的斜杠。由于我在Windows 上写这个例子， 这个斜杠是一个反斜杠而不是正斜杠。如果你在Linux 或者Mac OS X上重现这个例子， 你将会看见正斜杠. 无论你使用哪种形式的斜杠，Python 都可以访问到文件。
3. `os.path.expanduser()` 用来将包含 `~` 符号（表示当前用户Home目录）的路径扩展为完整的路径。在任何有Home 目录概念的操作系统上(包括Linux，Mac OS X 和Windows)，这个函数都能工作。返回的路径不以斜杠结尾，但是 `os.path.join()` 并不介意这一点。
4. 结合这些技术，你可以很方便的构造出用户Home 目录下的文件和目录的路径。 `os.path.join()` 可以接受任何数量的参数。当我发现这一点时我大喜过望， 因为在一门新的语言中构造我的工具箱时， `addSlashIfNecessary()` 总是我不得不写的愚蠢的小函数之一。不要 在Python 中写这个愚蠢的小函数，聪明的人们已经帮你考虑过这个问题了。

`os.path` 也包含用于分割完整路径名，目录名和文件名的函数

```
>>> pathname = '/Users/pilgrim/diveintopython3/examples/humansize.py'

('/Users/pilgrim/diveintopython3/examples', 'humansize.py')

'/Users/pilgrim/diveintopython3/examples'

'humansize.py'

>>> shortname
'humansize'
>>> extension
'.py'
```

1. `split` 函数分割一个完整路径并返回目录和文件名。
2. 还记得我说过在函数返回多个值时应该使用多变量赋值 吗？ `os.path.split()` 函数正是这样做的。 将 `split` 函数的返回值赋值给一个二元组。每个变量获得了返回元组中的对应元素的值。
3. 第一个变量 `dirname` ， 获得了 `os.path.split()` 函数返回元组中的第一个元素，文件所在的目录。
4. 第二个变量 `filename` ， 获得了 `os.path.split()` 函数返回元组中的第二个元素，文件名。
5. `os.path` 也包含 `os.path.splitext()` 函数，它分割一个文件名并返回短文件名和扩展名。可以使用同样的技术将它们的值赋值给不同的变量。

罗列目录内容

`glob` 模块是Python标准库中的另一个工具，它可以通过编程的方法获得一个目录的内容，并且它使用熟悉的命令行下的通配符。

`glob` 模块使用shell风格的通配符。

```
>>> os.chdir('/Users/pilgrim/diveintopython3/')
>>> import glob

['examples\\feed-broken.xml',
'examples\\feed-ns0.xml',
'examples\\feed.xml']

['alphameticstest.py',
'pluraltest1.py',
'pluraltest2.py',
'pluraltest3.py',
'pluraltest4.py',
'pluraltest5.py',
'pluraltest6.py',
'romantest1.py',
'romantest10.py',
'romantest2.py',
'romantest3.py',
'romantest4.py',
'romantest5.py',
'romantest6.py',
'romantest7.py',
'romantest8.py',
'romantest9.py']
```

1. `glob` 模块接受一个通配符并返回所有匹配的文件和目录的路径。在这个例子中，通配符是一个目录名加上 “ `*.xml` ”， 它匹配 `examples` 子目录下的所有 `.xml` 文件。
2. 现在我们将当前工作目录切换到 `examples` 目录。 `os.chdir()` 可以接受相对路径.
3. 在glob模式中你可以使用多个通配符。这个例子在当前工作目录中找出所有扩展名为 `.py` 并且在文件名中包含单词 `test` 的文件。

获取文件元信息

每一个现代文件系统都对文件存储了元信息: 创建时间，最后修改时间，文件大小等等。Python 单独提供了一个的API 用于访问这些元信息。 你不需要打开文件。知道文件名就足够了。

```
>>> import os

c:\Users\pilgrim\diveintopython3\examples

1247520344.9537716

time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
tm_min=25, tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

1. 当前工作目录是 `examples` 文件夹。
2. `feed.xml` 是 `examples` 文件夹中的一个文件。 调用 `os.stat()` 函数返回一个包含多种文件元信息的对象。
3. `st_mtime` 是最后修改时间，它的格式不是很有用。(技术上讲，它是从纪元，也就是1970年1月1号的第一秒钟，到现在的秒数)
4. `time` 模块是Python标准库的一部分。 它包含用于在不同时间格式中转换，将时间格式化成字符串以及处理时区的函数。
5. `time.localtime()` 函数将从纪元到现在的秒数这个格式表示的时间(`os.stat()` 函数返回值的 `st_mtime` 属性)转换成更有用的包含年、月、日、小时、分钟、秒的结构体。这个文件的最后修改时间是2009年7月13日下午5:25。

```
# continued from the previous example

3070
>>> import humansize

'3.0 KiB'
```

1. `os.stat()` 函数也通过 `st_size` 属性返回文件大小。文件 `feed.xml` 的大小是 `3070` 字节。
2. 你可以将 `st_size` 属性作为参数传给 `approximate_size()` 函数。

构造绝对路径

在前一节中， `glob.glob()` 函数返回一个相对路径的列表。第一个例子的路径类似 `'examples\feed.xml'` ，而第二个例子的路径 `'romantest1.py'` 更短。只要你保持在当前工作目录中，你就可以使用这些相对路径来打开文件或者获得文件的元信息。但是当你希望构造一个从根目录开始或者是包含盘符的绝对路径时，你就需要用到 `os.path.realpath()` 函数了。

```
>>> import os
>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\examples
>>> print(os.path.realpath('feed.xml'))
c:\Users\pilgrim\diveintopython3\examples\feed.xml
```

列表解析

你可以在列表解析中使用任何的Python表达式。

列表解析提供了一种紧凑的方式，实现了通过对列表中每一个元素应用一个函数的方法来将一个列表映射到另一个列表.

```
>>> a_list = [1, 9, 8, 4]

[2, 18, 16, 8]

[1, 9, 8, 4]

>>> a_list
[2, 18, 16, 8]
```

1. 为了理解这一点，请从右向左看。 `a_list` 是你要映射的列表。Python解释器逐个访问 `a_list` 的元素，并临时将元素赋值给变量 `elem` 。 然后Python 对元素应用函数 `` `elem `` * 2`并且将结果添加到返回列表中。
2. 列表解析创造一个新的列表而不改变原列表。
3. 可以安全的将列表解析的结果赋值给被映射的变量。Python会在内存中构造新的列表，在列表解析完成后将结果赋值给原来的变量。

你可以在列表解析中使用任何的Python表达式， 包括 `os` 模块中用于操作文件和目录的函数。

```
>>> import os, glob

['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']

['c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml',
'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml',
'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml']
```

1. 这里返回当前目录下的所有 `.xml` 文件。
2. 列表解析接受 `.xml` 文件列表并将其转化成全路径的列表。

列表解析也可以过滤列表，生成比原列表短的结果列表。

```
>>> import os, glob

['pluraltest6.py',
'romantest10.py',
'romantest6.py',
'romantest7.py',
'romantest8.py',
'romantest9.py']
```

1. 你可以在列表解析的最后加入 `if` 子句来过滤列表。对于列表中每一个元素 `if` 关键字后面的表达式都会被计算。如果表达式的计算结果为 `True`，那么这个元素将会被包含在输出中。这个列表解析在当前目录查找所有 `.py` 文件，而 `if` 表达式通过测试文件大小是否大于 `6000` 字节对列表进行过滤。有6个符合条件的文件，所以这个列表解析返回包含六个文件名的列表。

到目前为止的例子中的列表解析都只是用了一些简单的表达式， 乘以一个常数、调用一个函数或者是在过滤后返回原始元素。 然而列表解析并不限制表达式的复杂程度。

```
>>> import os, glob

[(3074, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml'),
(3386, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml'),
(3070, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml')]
>>> import humansize

[('3.0 KiB', 'feed-broken.xml'),
('3.3 KiB', 'feed-ns0.xml'),
('3.0 KiB', 'feed.xml')]
```

1. 这个列表解析找到当前工作目录下的所有 `.xml` 文件， 对于每一个文件构造一个包含文件大小(通过调用 `os.stat()` 获得)和绝对路径(通过调用 `os.path.realpath()`)的元组。
2. 这个列表解析在前一个的基础上对每一个 `.xml` 文件的大小应用 `approximate_size()` 函数。

字典解析

字典解析和列表解析类似，只不过它生成字典而不是列表。

```
>>> import os, glob

('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
st_mtime=1247520344, st_ctime=1247520344))

<class 'dict'>

['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
'romantest9.py', 'pluraltest3.py', 'romantest1.py', 'romantest2.py',
'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
'pluraltest4.py']

2509
```

1. 这不是字典解析; 而是列表解析。它找到所有名称中包含 `test` 的 `.py` 文件，然后构造包含文件名和文件元信息(通过调用 `os.stat()` 函数得到)的元组。
2. 结果列表的每一个元素是元组。
3. 这是一个字典解析。 除了两点以外，它的语法同列表解析很类似。首先，它被花括号而不是方括号包围; 第二，对于每一个元素它包含由冒号分隔的两个表达式，而不是列表解析的一个。冒号前的表达式(在这个例子中是 `f`)是字典的键;冒号后面的表达式(在这个例子中是 `os.stat(f)`)是值。
4. 字典解析返回结果是字典。
5. 这个字典的键很简单，就是 `glob.glob('*test*.py')` 调用返回的文件名。
6. 每一个键对应的值是 `os.stat()` 函数的返回值。这意味着我们可以在字典中通过文件名查找到它的文件元信息。元信息的一个部分是文件大小 `st_size` 。这个文件 `alphameticstest.py` 的大小是 `2509` 字节。

同列表解析一样，你可以在字典解析中包含 `if` 字句来过滤输入序列，对于每一个元素字句中的表达式都会被求值。

```
>>> import os, glob, humansize

['romantest9', 'romantest8', 'romantest7', 'romantest6', 'romantest10', 'pluraltest6']

'6.5 KiB'
```

1. 这个字典解析获得当前目录下所有的文件的列表(`glob.glob('*')`)，通过 `os.stat(f)` 获得每一个文件的元信息， 然后构造一个键是文件名，值是文件元信息的字典。
2. 这个字典解析在前一个基础上过滤掉文件小于 `6000` 字节的文件(`if meta.st_size > 6000`)， 并用过滤出的列表构造字典， 字典的键是文件名去掉扩展名的部分(`os.path.splitext(f)[0]`)，字典的值是每个文件的人类可读的近似大小(`humansize.approximate_size(meta.st_size)`)。
3. 正如你在前一个例子中所看见的，有6个这样的文件，所以字典中有6个元素。
4. 每一个键对应的值是 `approximate_size()` 函数返回的字符串。

其他同字典解析有关的小技巧

这里是一个可能有用的通过字典解析实现的小技巧: 交换字典的键和值。

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

集合解析

同样,集合也有自己的集合解析的语法。它和字典解析的非常相似，唯一的不同是集合只有值而没有键:值对。

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

{0, 8, 2, 4, 6}

{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```

1. 集合解析可以接受一个集合作为参数。这个集合解析计算数字0- `9` 这个集合的的平方。
2. 同列表解析和字典解析一样， 集合解析也可以包含 `if` 字句来在将元素放入结果集合前进行过滤。
3. 集合解析的输入并不一定要是集合; 可以是任何序列。

进一步阅读

- `os` module
- `os` — Portable access to operating system specific features
- `os.path` module
- `os.path` — Platform-independent manipulation of file names
- `glob` module
- `glob` — Filename pattern matching
- `time` module
- `time` — Functions for manipulating clock time
- List comprehensions
- Nested list comprehensions
- Looping techniques

Chapter 4 字符串

> " I'm telling you this 'cause you're one of my friends. My alphabet starts where your alphabet ends! " — Dr. Seuss, On Beyond Zebra!

在开始之前需要掌握的一些知识

你是否知道 Bougainville 人有世界上最小的字母表？他们的 Rotokas 字母表只包含了12个字母： A, E, G, I, K, O, P, R, S, T, U, 和 V。另一方面，像汉语，日语和韩语这些语言，它们则有成千上万个字符。当然啦，英语共有26个字母 — 如果把大写和小写分别计算的话，52个 — 外加少量的标点符号，比如*!@#$%&*

当人们说起"文本"，他们通常指显示在屏幕上的字符或者其他的记号；但是计算机不能直接处理这些字符和标记；它们只认识位(bit)和字节(byte)。实际上，从屏幕上的每一块文本都是以某种字符编码*(character encoding)*的方式保存的。粗略地说就是，字符编码提供一种映射，使屏幕上显示的内容和内存、磁盘内存储的内容对应起来。有许多种不同的字符编码，有一些是为特定的语言，比如俄语、中文或者英语，设计、优化的，另外一些则可以用于多种语言的编码。

在实际操作中则会比上边描述的更复杂一些。许多字符在几种编码里是共用的，但是在实际的内存或者磁盘上，不同的编码方式可能会使用不同的字节序列来存储他们。所以，你可以把字符编码当做一种解码密钥。当有人给你一个字节序列 — 文件，网页，或者别的什么 — 并且告诉你它们是"文本"时，就需要知道他们使用了何种编码方式，然后才能将这些字节序列解码成字符。如果他们给的是错误的"密钥"或者根本没有给你"密钥"，那就得自己来破解这段编码，这可是一个艰难的任务。有可能你使用了错误的解码方式，然后出现一些莫名其妙的结果。

你所了解的关于字符串的知识都是错的。

你肯定见过这样的网页，在撇号(')该出现的地方被奇怪的像问号的字符替代了。这种情况通常意味着页面的作者没有正确的声明其使用的编码方式，浏览器只能自己来猜测，结果就是一些正确的和意料之外的字符的混合体。如果原文是英语，那只是不方便阅读而已；在其他的语言环境下，结果可能是完全不可读的。

现有的字符编码各类给世界上每种主要的语言都提供了编码方案。由于每种语言的各不相同，而且在以前内存和硬盘都很昂贵，所以每种字符编码都为特定的语言做了优化。上边这句话的意思是，每种编码都使用数字(0–255)来代表这种语言的字符。比如，你也许熟悉ASCII编码，它将英语中的字符都当做从0–127的数字来存储。（65表示大写的"A"，97表示小写的"a"，_&_c。）英语的字母表很简单，所以它能用不到128个数字表达出来。如果你懂得2进制计数的话，它只使用了一个字节内的7位。

西欧的一些语言，比如法语，西班牙语和德语等，比英语有更多的字母。或者，更准确的说，这些语言含有与变音符号(diacritical marks)组合起来的字母，像西班牙语里的 `ñ`。这些语言最常用的编码方式是CP-1252，又叫做“windows-1252”，因为它在微软的视窗操作系统上被广泛使用。CP-1252和ASCII在0–127这个范围内的字符是一样的，但是CP-1252为 `ñ`(n-with-a-tilde-over-it, 241)， `ü`(u-with-two-dots-over-it, 252)这类字符而扩展到了128–255这个范围。然而，它仍然是一种单字节的编码方式；可能的最大数字为255，这仍然可以用一个字节来表示。

然而，像中文，日语和韩语等语言，他们的字符如此之多而不得不需要多字节编码的字符集。即，使用两个字节的数字(0–255)代表每个“字符”。但是就跟不同的单字节编码方式一样，多字节编码方式之间也有同样的问题，即他们使用的数字是相同的，但是表达的内容却不同。相对于单字节编码方式它们只是使用的数字范围更广一些，因为有更多的字符需要表示。

在没有网络的时代，“文本”由自己输入，偶尔才会打印出来，大多数情况下使用以上的编码方案是可行的。那时没有太多的“纯文本”。源代码使用ASCII编码，其他人也都使用字处理器，这些字处理器定义了他们自己的格式（非文本的），这些格式会连同字符编码信息和风格样式一起记录其中，_&_c。人们使用与原作者相同的字处理软件读取这些文档，所以或多或少地能够使用。

现在，我们考虑一下像email和web这样的全球网络的出现。大量的“纯文本”文件在全球范围内流转，它们在一台电脑上被撰写出来，通过第二台电脑进行传输，最后在另外一台电脑上显示。计算机只能识别数字，但是这些数字可能表达的是其他的东西。Oh no! 怎么办呢。。好吧，那么系统必须被设计成在每一段“纯文本”上都搭载编码信息。记住，编码方式是将计算机可读的数字映射成人类可读的字符的解码密钥。失去解码密钥则意味着混乱不清的，莫名其妙的信息，或者更糟。

现在我们考虑尝试把多段文本存储在同一个地方，比如放置所有收到邮件的数据库。这仍然需要对每段文本存储其相关的字符编码信息，只有这样才能正确地显示它们。这很困难吗？试试搜索你的email数据库，这意味着需要在运行时进行编码之间的转换。很有趣是吧…

现在我们来分析另外一种可能性，即多语言文档，同一篇文档里来自几种不同语言的字符混在一起。（提示：处理这样文档的程序通常使用转义符在不同的“模式(modes)”之间切换。噗！现在是俄语 koi8-r 模式，所以241代表 Я；噗噗！现在到了Mac Greek模式，所以241代表 ώ。）当然，你也会想要搜索这些文档。

现在，你就哭吧，因为以前所了解的关于字符串的知识都是错的，根本就没有所谓的“纯文本”。

Unicode

*Unicode*入门。

Unicode编码系统为表达任意语言的任意字符而设计。它使用4字节的数字来表达每个字母、符号，或者表意文字(ideograph)。每个数字代表唯一的至少在某种语言中使用的符号。（并不是所有的数字都用上了，但是总数已经超过了65535，所以2个字节的数字是不够用的。）被几种语言共用的字符通常使用相同的数字来编码，除非存在一个在理的语源学(etymological)理由使不这样做。不考虑这种情况的话，每个字符对应一个数字，每个数字对应一个字符。即不存在二义性。不再需要记录“模式”了。 `U+0041` 总是代表 `'A'` ，即使这种语言没有 `'A'` 这个字符。

初次面对这个创想，它看起来似乎很伟大。一种编码方式即可解决所有问题。文档可包含多种语言。不再需要在各种编码方式之间进行“模式转换“。但是很快，一个明显的问题跳到我们面前。4个字节？只为了单独一个字符? 这似乎太浪费了，特别是对像英语和西语这样的语言，他们只需要不到1个字节即可以表达所需的字符。事实上，对于以象形为基础的语言（比如中文）这种方法也有浪费，因为这些语言的字符也从来不需要超过2个字节即可表达。

有一种Unicode编码方式每1个字符使用4个字节。它叫做UTF-82，因为32位 = 4字节。UTF-32是一种直观的编码方式；它收录每一个Unicode字符（4字节数字）然后就以那个数字代表该字符。这种方法有其优点，最重要的一点就是可以在常数时间内定位字符串里的第 `N` 个字符，因为第 `N` 个字符从第 `4×Nth` 个字节开始。另外，它也有其缺点，最明显的就是它使用4个“诡异”的字节来存储每个“诡异”的字符…

尽管有Unicode字符非常多，但是实际上大多数人不会用到超过前65535个以外的字符。因此，就有了另外一种Unicode编码方式，叫做UTF-16(因为16位 = 2字节)。UTF-16将0–65535范围内的字符编码成2个字节，如果真的需要表达那些很少使用的“星芒层(astral plane)”内超过这65535范围的Unicode字符，则需要使用一些诡异的技巧来实现。UTF-16编码最明显的优点是它在空间效率上比UTF-32高两倍，因为每个字符只需要2个字节来存储（除去65535范围以外的），而不是UTF-32中的4个字节。并且，如果我们假设某个字符串不包含任何星芒层中的字符，那么我们依然可以在常数时间内找到其中的第 `N` 个字符，直到它不成立为止这总是一个不错的推断…

但是对于UTF-32和UTF-16编码方式还有一些其他不明显的缺点。不同的计算机系统会以不同的顺序保存字节。这意味着字符 `U+4E2D` 在UTF-16编码方式下可能被保存为 `4E 2D` 或者 `2D 4E` ，这取决于该系统使用的是大尾端(big-endian)还是小尾端(little-endian)。（对于UTF-32编码方式，则有更多种可能的字节排列。）只要文档没有离开你的计算机，它还是安全的 — 同一台电脑上的不同程序使用相同的字节顺序(byte order)。但是当我们需要在系统之间传输这个文档的时候，也许在万维网中，我们就需要一种方法来指示当前我们的字节是怎样存储的。不然的话，接收文档的计算机就无法知道这两个字节 `4E 2D` 表达的到底是 `U+4E2D` 还是 `U+2D4E` 。

为了解决这个问题，多字节的Unicode编码方式定义了一个“字节顺序标记(Byte Order Mark)”，它是一个特殊的非打印字符，你可以把它包含在文档的开头来指示你所使用的字节顺序。对于UTF-16，字节顺序标记是 `U+FEFF` 。如果收到一个以字节 `FF FE` 开头的UTF-16编码的文档，你就能确定它的字节顺序是单向的(one way)的了；如果它以 `FE FF` 开头，则可以确定字节顺序反向了。

不过，UTF-16还不够完美，特别是要处理许多ASCII字符时。如果仔细想想的话，甚至一个中文网页也会包含许多的ASCII字符 — 所有包围在可打印中文字符周围的元素(element)和属性(attribute)。能够在常数时间内找到第 `Nth` 个字符当然非常好，但是依然存在着纠缠不休的星芒层字符的问题，这意味着你不能保证每个字符都是2个字节长，所以，除非你维护着另外一个索引，不然就不能真正意义上的在常数时间内定位第 `N` 个字符。另外，朋友，世界上肯定还存在很多的ASCII文本…

另外一些人琢磨着这些问题，他们找到了一种解决方法：

UTF-8 The range of integers used to code the abstract characters is called the codespace. A particular integer in this set is called a code point. When an abstract character is mapped or assigned to a particular code point in the codespace, it is then referred to as an encoded character. <-->

UTF-8是一种为Unicode设计的变长*(variable-length)*编码系统。即，不同的字符可使用不同数量的字节编码。对于ASCII字符(A-Z, _&_c.)UTF-8仅使用1个字节来编码。事实上，UTF-8中前128个字符(0–127)使用的是跟ASCII一样的编码方式。像ñ和ö这样的"扩展拉丁字符(Extended Latin)"则使用2个字节来编码。（这里的字节并不是像UTF-16中那样简单的Unicode编码点(unicode code point)；它使用了一些位变换(bit-twiddling)。）中文字符比如"中"则占用了3个字节。很少使用的"星芒层字符"则占用4个字节。

缺点：因为每个字符使用不同数量的字节编码，所以寻找串中第 `N` 个字符是一个O(N)复杂度的操作 — 即，串越长，则需要更多的时间来定位特定的字符。同时，还需要位变换来把字符编码成字节，把字节解码成字符。

优点：在处理经常会用到的ASCII字符方面非常有效。在处理扩展的拉丁字符集方面也不比UTF-16差。对于中文字符来说，比UTF-32要好。同时，（在这一条上你得相信我，因为我不打算给你展示它的数学原理。）由位操作的天性使然，使用UTF-8不再存在字节顺序的问题了。一份以UTF-8编码的文档在不同的计算机之间是一样的比特流。

概述

在Python 3，所有的字符串都是使用Unicode编码的字符序列。不再存在以UTF-8或者CP-1252编码的情况。也就是说，"这个字符串是以UTF-8编码的吗？不再是一个有效问题。"UTF-8是一种将字符编码成字节序列的方式。如果需要将字符串转换成特定编码的字节序列，Python 3可以为你做到。如果需要将一个字节序列转换成字符串，Python 3也能为你做到。字节即字节，并非字符。字符在计算机内只是一种抽象。字符串则是一种抽象的序列。

```
9
'深'
'深入 Python 3'
```

1. 为了创建一个字符串，将其用引号包围。Python字符串可以通过单引号(`'`)或者双引号(`"`)来定义。
2. 内置函数 `len()` 可返回字符串的长度，即字符的个数。这与获得列表，元组，集合或者字典的长度的函数是同一个。Python中，字符串可以想像成由字符组成的元组。
3. Just like getting individual items out of a list, you can get individual characters out of a string using index notation. 与取得列表中的元素一样，也可以通过下标记号取得字符串中的某个字符。
4. 类似列表，可以使用 `+` 操作符来连接(concatenate)字符串。

格式化字符串

字符串可以使用单引号或者双引号来定义。

我们再来看一看 `humansize.py` ：

```
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    if size < 0:

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:

    raise ValueError('number too large')
```

1. `'KB'` , `'MB'` , `'GB'` … 这些是字符串。
2. 函数的文档字符串(docstring)也是字符串。当前的文档字符串占用了多行，所以它使用了相邻的3个引号来标记字符串的起始和终止。
3. 这3个引号代表该文档字符串的终止。
4. 这是另外一个字符串，作为一个可读的提示信息传递给异常。
5. 瓦哦…那是什么？

Python 3支持把值格式化(format)成字符串。可以有非常复杂的表达式，最基本的用法是使用单个占位符(placeholder)将一个值插入字符串。

```
>>> username = 'mark'

"mark's password is PapayaWhip"
```

1. 不，`PapayaWhip` 真的不是我的密码。
2. 这里包含了很多知识。首先，这里使用了一个字符串字面值的方法调用。字符串也是对象，对象则有其方法。其次，整个表达式返回一个字符串。最后，`{0}` 和 `{1}` 叫做替换字段*(replacement field)*，他们会被传递给 `format()` 方法的参数替换。

复合字段名

在前一个例子中，替换字段只是简单的整数，这是最简单的用法。整型替换字段被当做传给 `format()` 方法的参数列表的位置索引。即，`{0}` 会被第一个参数替换（在此例中即 `username` ），`{1}` 被第二个参数替换（ `password`），_&_c。可以有跟参数一样多的替换字段，同时你也可以使用任意多个参数来调用 format() 。但是替换

```
>>> import humansize

>>> si_suffixes
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']

'1000KB = 1MB'
```

1. 不需要调用 `humansize` 模块定义的任何函数我们就可以抓取到其所定义的数据结构：国际单位制(SI, 来自法语Système International)的后缀列表（以1000为进制）。
2. 这一句看上去有些复杂，其实不是这样的。`{0}` 代表传递给 `format()` 方法的第一个参数，即 `si_suffixes` 。注意 `si_suffixes` 是一个列表。所以 `{0[0]}` 指代 `si_suffixes` 的第一个元素，即 `'KB'` 。同时，`{0[1]}` 指代该列表的第二个元素，即：`'MB'` 。大括号以外的内容 — 包括 `1000` ，等号，还有空格等 — 则按原样输出。语句最后返回字符串为 `'1000KB = 1MB'` 。

{0}会被format()的第1个参数替换，{1}则被其第2个参数替换。

这个例子说明格式说明符可以通过利用（类似）*Python*的语法访问到对象的元素或属性。这就叫做复合字段名*(compound field names)*。以下复合字段名都是“有效的”。

- 使用列表作为参数，并且通过下标索引来访问其元素（跟上一例类似）
- 使用字典作为参数，并且通过键来访问其值
- 使用模块作为参数，并且通过名字来访问其变量及函数
- 使用类的实例作为参数，并且通过名字来访问其方法和属性
- 以上方法的任意组合

为了使你确信的确如此，下面这个样例就组合使用了上面所有方法：

```
>>> import humansize
>>> import sys
>>> '1MB = 1000{0.modules[humansize].SUFFIXES[1000][0]}'.format(sys)
'1MB = 1000KB'
```

下面是描述它如何工作的：

- `sys` 模块保存了当前正在运行的Python实例的信息。由于已经导入了这个模块，因此可以将其作为 `format()` 方法的参数。所以替换域 `{0}` 指代 `sys` 模块。
- `sys.modules` is a dictionary of all the modules that have been imported in this Python instance. The keys are the module names as strings; the values are the module objects themselves. So the replacement field `{0.modules}` refers to the dictionary of imported modules. `sys.modules` 是一个保存当前Python实例中所有已经导入模块的字典。模块的名字作为字典的键；模块自身则是键所对应的值。所以 `{0.modules}` 指代保存当前已被导入模块的字典。
- `sys.modules['humansize']` 即刚才导入的 `humansize` 模块。所以替换域 `{0.modules[humansize]}` 指代 `humansize` 模块。请注意以上两句在语法上轻微的不同。在实际的Python代码中，字典 `sys.modules` 的键是字符串类型的；为了引用它们，我们需要在模块名周围放上引号（比如 `'humansize'` ）。但是在使用替换域的时候，我们在省略了字典的键名周围的引号（比如 `humansize` ）。在此，我们引用PEP 3101：字符串格式化高级用法，"解析键名的规则非常简单。如果名字以数字开头，则它被当作数字使用，其他情况则被认为是字符串。"
- `sys.modules['humansize'].SUFFIXES` 是在 `humansize` 模块的开头定义的一个字典对象。`{0.modules[humansize].SUFFIXES}` 即指向该字典。
- `sys.modules['humansize'].SUFFIXES[1000]` 是一个SI（国际单位制）后缀列表：`['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']` 。所以替换域 `{0.modules[humansize].SUFFIXES[1000]}` 指向该列表。
- `sys.modules['humansize'].SUFFIXES[1000][0]` 即SI后缀列表的第一个元素：`'KB'` 。因此，整个替换域 `{0.modules[humansize].SUFFIXES[1000][0]}` 最后都被两个字符 `KB` 替换。

格式说明符

但是，还有一些问题我们没有讲到！再来看一看 `humansize.py` 中那一行奇怪的代码：

```
if size < multiple:
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` 会被传递给 `format()` 方法的第二个参数替换，即 `suffix` 。但是 `{0:.1f}` 是什么意思呢？它其实包含了两方面的内容：`{0}` 你已经能理解，`:.1f` 则不一定了。第二部分（包括冒号及其后边的部分）即格式说明符*(format specifier)*，它进一步定义了被替换的变量应该如何被格式化。

> ☞格式说明符的允许你使用各种各种实用的方法来修饰被替换的文本，就像C语言中的 `printf()` 函数一样。我们可以添加使用零填充(zero-padding)，衬距(space-padding)，对齐字符串(align strings)，控制10进制数输出精度，甚至将数字转换成16进制数输出。

在替换域中，冒号(`:`)标示格式说明符的开始。“ `.1` ”的意思是四舍五入到保留一们小数点。“ `f` ”的意思是定点数（与指数标记法或者其他10进制数表示方法相对应）。因此，如果给定 `size为` 698.24 ， suffix 为 'GB' ，那么格式化后的字符串将是 '698.2 GB' ，因为 698.24 被四舍五入到一位小数表示，然后后缀 'GB' 再被追加到这个串最后。

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.2 GB'
```

想了解格式说明符的复杂细节，请参阅Python官方文档关于格式化规范的迷你语言

其他常用字符串方法

除了格式化，关于字符串还有许多其他实用的使用技巧。

```
... sult of years of scientif-
... ic study combined with the
... experience of years.'''

['Finished files are the re-',
 'sult of years of scientif-',
 'ic study combined with the',
 'experience of years.']

finished files are the re-
sult of years of scientif-
ic study combined with the
experience of years.

6
```

1. 我们可以在Python的交互式shell里输入多行(multiline)字符串。一旦我们以三个引号标记多行字符串的开始，按 `ENTER` 键，Python shell会提示你继续这个字符串的输入。连续输入三个结束引号以终止该字符串的输入，再敲 `ENTER` 键则会执行该条命令（在当前例子中，把这个字符串赋给变量 `s` ）。
2. `splitlines()` 方法以多行字符串作为输入，返回一个由字符串组成的列表，列表的元素即原来的单行字符串。请注意，每行行末的回车符没有被包括进去。
3. `lower()` 方法把整个字符串转换成小写的。（类似地， `upper()` 方法执行大写化转换操作。）
4. `count()` 方法对串中的指定的子串进行计数。是的，在那一句中确实出现了6个字母“f”。

还有一种经常会遇到的情况。比如有如下形式的键-值对列表 `` key1`=`value1`&`key2`=`value2 `` ，我们需要将其分离然后产生一个这样形式的字典 `{key1: value1, key2: value2}` 。

```
>>> query = 'user=pilgrim&database=master&password=PapayaWhip'

>>> a_list
['user=pilgrim', 'database=master', 'password=PapayaWhip']

>>> a_list_of_lists
[['user', 'pilgrim'], ['database', 'master'], ['password', 'PapayaWhip']]

>>> a_dict
{'password': 'PapayaWhip', 'user': 'pilgrim', 'database': 'master'}
```

1. `split()` 方法使用一个参数，即指定的分隔符，然后根据这个分隔符将串分离成一个字符串列表。此处，分隔符即字符“ `&` ”，它还可以是其他的内容。
2. 现在我们有了一个字符串列表，其中的每个串由三部分组成：键，等号和值。我们可以使用列表解析来遍历整个列表，然后利用第一个等号标记将每个字符串再分离成两个子串。（理论上，值也可以包含等号标记，如果执行 `'key=value=foo'.split('=')` ，那么我们会得到一个三元素列表 `['key', 'value', 'foo']` 。）
3. 最后，通过调用 `dict()` 函数Python会把那个包含列表的列表(list-of-lists)转换成字典对象。

> ☞上一个例子跟解析URL的请求参数(query parameters)很相似，但是真实的URL解析实际上比这个复杂得多。如果需要处理URL请求参数，我们最好使用 `urllib.parse.parse_qs()` 函数，它可以处理一些不常见的边缘情况。

字符串的分片

定义一个字符串以后，我们可以截取其中的任意部分形成新串。这种操作被称作字符串的分片(*slice*)。字符串分片跟列表的分片(slicing lists)原理是一样的，从直观上也说得通，因为字符串本身就是一些字符序列。

```
>>> a_string = 'My alphabet starts where your alphabet ends.'

'alphabet'

'alphabet starts where your alphabet en'

'My'

'My alphabet starts'

' where your alphabet ends.'
```

1. 我们可以通过指定两个索引值来获得原字符串的一个“slice”。该操作的返回值是一个新串，依次包含了从原串中第一个索引位置开始，直到但是不包含第二个索引位置之间的所有字符。
2. 就像给列表做分片一样，我们也可以使用负的索引值来分片字符串。
3. 字符串的下标索引是从0开始的，所以 `a_string[0:2]` 会返回原字符串的前两个元素，从 `a_string[0]` 开始，直到但不包括 `a_string[2]` 。
4. 如果省略了第一个索引值，Python会默认它的值为0。所

以 `a_string[:18]` 跟 `a_string[0:18]` 的效果是一样的，因为从0开始是被Python默认的。

5. 同样地，如果第2个索引值是原字符串的长度，那么我们也可以省略它。所以，在此处 `a_string[18:]` 跟 `a_string[18:44]` 的结果是一样的，因为这个串的刚好有44个字符。这种规则存在某种有趣的对称性。在这个由44个字符组成的串中，`a_string[:18]` 会返回前18个字符，而 `a_string[18:]` 则会返回除了前18个字符以外字符串的剩余部分。事实上 `a_string[: n ]` 总是会返回串的前 `n` 个字符，而 `a_string[ n :]` 则会返回其余的部分，这与串的长度无关。

String vs. Bytes

字节即字节；字符是一种抽象。一个不可变(immutable)的Unicode编码的字符序列叫做*string*。一串由0到255之间的数字组成的序列叫做*bytes*对象。

```
>>> by
b'abcde'

<class 'bytes'>

5

>>> by
b'abcde\xff'

6

97

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

1. 使用“byte字面值”语法 `b''` 来定义 `bytes` 对象。byte字面值里的每个字节可以是ASCII字符或者是从 `\x00` 到 `\xff` 编码了的16进制数。
2. `bytes` 对象的类型是 `bytes` 。
3. 跟列表和字符串一样，我们可以通过内置函数 `len()` 来获得 `bytes` 对象的长度。
4. 使用 `+` 操作符可以连接 `bytes` 对象。操作的结果是一个新的 `bytes` 对象。
5. 连接5个字节的和1个字节的 `bytes` 对象会返回一个6字节的 `bytes` 对象。
6. 一如列表和字符串，可以使用下标记号来获取 `bytes` 对象中的单个字节。对字符串做这种操作获得的元素仍为字符串，而对 `bytes` 对象做这种操作的返回值则为整数。确切地说，是0–255之间的整数。
7. `bytes` 对象是不可变的；我们不可以给单个字节赋上新值。如果需要改变某个字节，可以组合使用字符串的切片和连接操作(效果跟字符串是一样的)，或者我们也可以将 `bytes` 对象转换为 `bytearray` 对象。

```
>>> by = b'abcd\x65'

>>> barr
bytearray(b'abcde')

5

>>> barr
bytearray(b'fbcde')
```

1. 使用内置函数 `bytearray()` 来完成从 `bytes` 对象到可变的 `bytearray` 对象的转换。
2. 所有对 `bytes` 对象的操作也可以用在 `bytearray` 对象上。
3. 有一点不同的就是，我们可以使用下标标记给 `bytearray` 对象的某个字节赋值。并且，这个值必须是0–255之间的一个整数。

我们决不应该这样混用bytes和strings。

```
>>> by = b'd'
>>> s = 'abcde'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly

1
```

1. 不能连接 `bytes` 对象和字符串。他们两种不同的数据类型。
2. 也不允许针对字符串中 `bytes` 对象的出现次数进行计数，因为串里面根本没有 `bytes` 。字符串是一系列的字符序列。也许你是想要先把这些字节序列通过某种编码方式进行解码获得字符串，然后对该字符串进行计数？可以，但是需要显式地指明它。Python 3不会隐含地将bytes转换成字符串，或者进行相反的操作。
3. 好巧啊…这一行代码刚好给我们演示了使用特定编码方式将 `bytes` 对象转换成字符串后该串的出现次数。

所以，这就是字符串与字节数组之间的联系了： `bytes` 对象有一个 `decode()` 方法，它使用某种字符编码作为参数，然后依照这种编码方式将 `bytes` 对象转换为字符串，对应地，字符串有一个 `encode()` 方法，它也使用某种字符编码作为参数，然后依照它将串转换为 `bytes` 对象。在上一个例子中，解码的过程相对直观一些 — 使用ASCII编码将一个字节序列转换为字符串。同样的过程对其他的编码方式依然有效 — 传统的（非Unicode）编码方式也可以，只要它们能够编码串中的所有字符。

```
>>> len(a_string)
9

>>> by
b'\xe6\xb7\xb1\xe5\x85\xa5 Python'
>>> len(by)
13

>>> by
b'\xc9\xee\xc8\xeb Python'
>>> len(by)
11

>>> by
b'\xb2`\xa4J Python'
>>> len(by)
11

>>> roundtrip
'深入 Python'
>>> a_string == roundtrip
True
```

1. `a_string` 是一个字符串。它有9个字符。
2. `by` 是一个 `bytes` 对象。它有13个字节。它是通过 `a_string` 使用UTF-8编码而得到的一串字节序列。
3. `by` 还是一个 `bytes` 对象。它有11个字节。它是通过 `a_string` 使用GB18030编码而得到的一串字节序列。
4. 此时的 `by` 仍旧是一个 `bytes` 对象，由11个字节组成。它又是一种完全不同的字节序列，我们通过对 `a_string` 使用Big5编码得到。
5. `roundtrip` 是一个字符串，共有9个字符。它是通过对 `by` 使用Big5解码算法得到的一个字符序列。并且，从执行结果可以看出， `roundtrip` 与 `a_string` 是完全一样的。

补充内容：Python源码的编码方式

Python 3会假定我们的源码 — 即 `.py` 文件 — 使用的是UTF-8编码方式。

> ☞Python 2里， `.py` 文件默认的编码方式为ASCII。Python 3的源码的默认编码方式为UTF-8

如果想使用一种不同的编码方式来保存Python代码，我们可以在每个文件的第一行放置编码声明(encoding declaration)。以下声明定义 `.py` 文件使用windows-1252编码方式：

```
# -*- coding: windows-1252 -*-
```

从技术上说，字符编码的重载声明也可以放在第二行，如果第一行被类UNIX系统中的hash-bang命令占用了。

```
#!/usr/bin/python3
# -*- coding: windows-1252 -*-
```

了解更多信息，请参阅PEP 263: 指定Python源码的编码方式。

进一步阅读

关于Python中的Unicode：

- Python Unicode HOWTO
- Python 3中的新鲜事: 文本 vs. 数据，而非 Unicode vs. 8-bit

关于Unicode本身：

- 每个软件开发人员应该无条件、至少掌握的关于Unicode和字符集的知识
- 关于Unicode的优势
- 关于字元字串(character string)
- 字符 vs. 字节

关于其他的编码方式：

- XML文档的编码方式
- HTML文档的编码方式

关于字符串及其格式化：

- `string` — 常用字符串操作
- 格式化字符串的语法
- 关于格式化规范的迷你语言
- PEP 3101: 字符串格式化高级应用

Updated October 7, 2009 • Difficulty level: ♦♦♦♢♢

Chapter 5 正则表达式

> " Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. " — Jamie Zawinski

深入

所有的现代编程语言都有内建字符串处理函数。在python里查找，替换字符串的方法是：index()、 find()、split()、 count()、 replace()等。但这些方法都只是最简单的字符串处理。比如：用index()方法查找单个子字符串，而且查找总是区分大小写的。为了使用不区分大小写的查找，可以使用s.lower()或者s.upper()，但要确认你查找的字符串的大小写是匹配的。replace() 和split() 方法有相同的限制。

如果使用string的方法就可以达到你的目的，那么你就使用它们。它们速度快又简单，并且很容易阅读。但是如果你发现自己要使用大量的if语句，以及很多字符串函数来处理一些特例，或者说你需要组合调用split() 和 join() 来切片、合并你的字符串，你就应该使用正则表达式。

正则表达式有强大并且标准化的方法来处理字符串查找、替换以及用复杂模式来解析文本。正则表达式的语法比我们的程序代码更紧凑，格式更严格，比用组合调用字符串处理函数的方法更具有可读性。甚至你可以在正则表达式中嵌入注释信息，这样就可以使它有自文档化的功能。

> ☞如果你在其他语言中使用过正则表达式（比如perl，javascript或者php），python的正则表达式语法和它们的很像。阅读re模块的摘要信息可以了解到一些处理函数以及它们参数的一些概况。

案例研究: 街道地址

下面一系列的示例的灵感来自于现实生活中我几年前每天的工作。我需要把一些街道地址导入一个新的系统，在这之前我要从一个遗留的老系统中清理和标准化这些街道地址。下面这个例子展示我怎么解决这个问题。

```
>>> s = '100 NORTH MAIN ROAD'

'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'

'100 NORTH BRD. RD.'

'100 NORTH BROAD RD.'

'100 NORTH BROAD RD.'
```

1. 我的目的是要标准化街道的格式。而‘ROAD’总是在.RD的前面。刚开始我以为只需要简单的使用string的replace()方法就可以。所有的数据都是大写的，因此不会出现大小写不匹配的问题。而查找的字符串‘ROAD’也是一个常量。在这个简单的例子中s.replace()可以很好的工作。
2. 事实上，不幸的是，我很快发现一个问题，在一些地址中‘ROAD’出现了两次，一个是前面的街道名里带了‘ROAD’，一个是‘ROAD’本身。repalce()发现了两个就把他们都给替换掉了。这意味着，我的地址错了。
3. 为了解决地址中出现超过一个‘ROAD’子字符串的问题，你可能会这么考虑：只在地址的最后四个字符中查找和替换“ROAD’（s[-4:]）。然后把剩下的字符串独立开来处理（s[:-4]）。这个方法很笨拙。比如，这个方法会依赖于你要替换的字符串长度（如果你用‘.ST’来替换‘STREET’，就需要在s[-6:]中查找‘STREET’，然后再取s[:-6]。你难道还想半年后回来继续修改BUG？反正我是不想。
4. 是时候转换到正则表达式了。在python中，所有的正则表达式相关功能都包含在re模块中。
5. 注意第一个参数‘ROAD$’，这是一个匹配‘ROAD’仅仅出现在字符串结尾的正则表达式。$表示“字符串结尾”。（还有一个相应的表示“字符串开头”的字符 ^ ）。正则表达式模块的re.sub()函数可以做字符串替换，它在字符串s中用正则表达式‘ROAD$’来搜索并替换成‘RD.’。它只会匹配字符串结尾的‘ROAD’，而不会匹配到‘BROAD’中的‘ROAD’，因为这种情况它在字符串的中间。

^ 匹配字符串开始. $ 匹配字符串结尾

继续我的处理街道地址的故事。我很快发现，在之前的例子中，匹配地址结尾的‘ROAD’不够好。因为并不是所有的地址结尾都有它。一些地址简单的用一个街道名结尾。大部分的情况下不会有问题，但如果街道的名字就叫‘BROAD’，这个时候，正则表达式会匹配到‘BROAD’的最后4个字符，这并不是我想要的。

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'

'100 BROAD'

'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'

'100 BROAD ROAD APT. 3'

'100 BROAD RD. APT 3'
```

1. 我真正想要的‘ROAD’，必须是匹配到字符串结尾，并且是独立的词（他不能是某个比较长的词的一部分）。为了在正则表达式中表达这个独立的词，你可以使用‘\b’。它的意思是“在右边必须有一个分隔符”。在python中，比较复杂的是‘\’字符必须被转义，这有的时候会导致‘\’字符传染（想想可能还要对\字符做转义的情况）。这也是为什么perl中的正则表达式比python的简单的原因之一。另一方面，perl会在正则表达式中混合其他非正则表达式的语法，如果出现了bug，那么很难区分这个bug是在正则表达式中，还是在其他的

语法部分。

2. 为了解决'\'字符传染的问题，可以使用原始字符串。这只需要在字符串的前面添加一个字符'r'。它告诉python，字符串中没有任何字符需要转义。'\t'是一个制表符，但r'\t'只是一个字符'\'紧跟着一个字符t。我建议在处理正则表达式的时候总是使用原始字符串。否则，会因为理解正则表达式而消耗大量时间（本身正则表达式就已经够让人困惑的了）。
3. 哎，不幸的是，我发现了更多的地方与我的逻辑背道而驰。街道地址包含了独立的单词'ROAD'，但并不是在字符串尾，因为街道后面还有个单元号。因为'ROAD'并不是最靠后，就不能匹配，因此re.sub()最后没有做任何的替换，只是返回了一个原始的字符串，这并不是你想要的。
4. 为了解决这个问题，我删除了正则表达式尾部的$，然后添加了一个\b。现在这个正则表达式的意思是"在字符串的任意位置匹配独立的'ROAD'单词"不管是在字符串的结束还是开始，或者中间的任意一个位置。

案例研究: 罗马数字

你肯定见过罗马数字，即使你不认识他们。你可能在版权信息、老电影、电视、大学或者图书馆的题词墙看到（用Copyright MCMXLVI" 表示版权信息，而不是用 "Copyright 1946"），你也可能在大纲或者目录参考中看到他们。这种系统的数字表达方式可以追溯到罗马帝国（因此而得名）。

在罗马数字中，有七个不同的数字可以以不同的方式结合起来表示其他数字。

- `I = 1`
- `V = 5`
- `X = 10`
- `L = 50`
- `C = 100`
- `D = 500`
- `M = 1000`

下面是几个通常的规则来构成罗马数字：

- 大部分时候用字符相叠加来表示数字。I是1， II是2， III是3。VI是6（挨个看来，是"5 和1"的组合），VII是7，VIII是8。
- 含有10的字符（I，X，C和M）最多可以重复出现三个。为了表示4，必须用同一位数的下一个更大的数字5来减去一。不能用IIII来表示4，而应该是IV（意思是比5小1）。40写做XL（比50小10），41写做XLI，42写做XLII，43写做XLIII，44写做XLIV（比50小10并且比5小1）。
- 有些时候表示方法恰恰相反。为了表示一个中间的数字，需要从一个最终的值来减。比如：9需要从10来减：8是VIII，但9确是IX（比10小1），并不是VIII（I字符不能重复4次）。90是XC，900是CM。

- 表示5的字符不能在一个数字中重复出现。10只能用X表示，不能用VV表示。100只能用C表示，而不是LL。
- 罗马数字是从左到右来计算，因此字符的顺序非常重要。DC表示600，而CD完全是另一个数字400（比500小100）。CI是101，IC不是一个罗马数字（因为你不能从100减1，你只能写成XCIX，表示比100小10，且比10小1）。

检查千位数

怎么验证一个字符串是否是一个合法的罗马数字呢？我们可以每次取一个字符来处理。因为罗马数字总是从高位到低位来书写。我们从最高位的千位开始。表示1000或者更高的位数值，方法是用一系列的M来重复表示。

```
>>> import re

<_sre.SRE_Match object at 0106FB58>

<_sre.SRE_Match object at 0106C290>

<_sre.SRE_Match object at 0106AA38>

<_sre.SRE_Match object at 0106F4A8>
```

1. 这个模式有三部分。^表示必须从字符串开头匹配。如果没有指定^，这个模式将在任意位置匹配M，这个可能并不是你想要的。你需要确认是否要匹配字符串开始的M，还是匹配单个M字符。因为它重复了三次，你要在一行中的任意位置匹配0到3次的M字符。$匹配字符串结束。当它和匹配字符串开始的^一起使用，表示匹配整个字符串。没有任何一个字符可在M的前面或者后面。
2. re模块最基本的方法是search()函数。它使用正则表达式来匹配字符串（M）。如果成功匹配，search()返回一个匹配对象。匹配对象中有很多的方法来描述这个匹配结果信息。如果没有匹配到，search()返回None。你只需要关注search()函数的返回值就可以知道是否匹配成功。‘M’被正则表达式匹配到了。原因是正则表达式中的第一个可选的M匹配成功，第二个和第三个被忽略掉了。
3. ‘MM’匹配成功。因为正则表达式中的第一个和第二个可选的M匹配到，第三个被忽略。
4. ‘MMM’匹配成功。因为正则表达式中的所有三个M都匹配到。
5. ‘MMMM’匹配失败。正则表达式中所有三个M都匹配到，接着正则表达式试图匹配字符串结束，这个时候失败了。因此search()函数返回None。
6. 有趣的是，空字符串也能匹配成功，因为正则表达式中的所有M都是可选的。

检查百位数

? 表示匹配是可选的

百位的匹配比千位复杂。根据值的不同，会有不同的表达方式。

- `100 = C`

- `200 = CC`
- `300 = CCC`
- `400 = CD`
- `500 = D`
- `600 = DC`
- `700 = DCC`
- `800 = DCCC`
- `900 = CM`

因此会有四种可能的匹配模式：

- `CM`
- `CD`
- 可能有0到3个字符C（0个表示千位为0）。
- D紧跟在0到3个字符C的后面。

这两个模式还可以组合起来表示：

- 一个可选的D，后面跟着0到3个字符C。

下面的例子展示了怎样在罗马数字中验证百位。

```
>>> import re
<_sre.SRE_Match object at 01070390>
<_sre.SRE_Match object at 01073A50>
<_sre.SRE_Match object at 010748A8>
<_sre.SRE_Match object at 01071D98>
```

1. 这个正则表达式的写法从上面千位的匹配方法接着往后写。检查字符串开始（^），然后是千位，后面才是新的部分。这里用圆括号定义了三个不同的匹配模式，他们是用竖线分隔的：CM，CD和D?C?C?C?（这表示是一个可选的D，以及紧跟的0到3个可选的字符C）。正则表达式按从左到右的顺序依次匹配，如果第一个CM匹配成功，用竖线分隔这几个中的后面其他的都会被忽略。
2. ‘MCM’匹配成功。因为第一个M匹配到，第二个和第三个M被忽略。后面的CM匹配到（因此后面的CD和D?C?C?C?根本就不被考虑匹配了）。MCM在罗马数字中表示1900。
3. ‘MD’匹配成功。因为第一个M匹配到，第二个和第三个M被忽略。然后D?C?C?C?匹配到D（后面的三个C都是可选匹配的，都被忽略掉）。MD在罗马数字中表示1500。
4. ‘MMMCCC’匹配成功。因为前面三个M都匹配到。后面的D?C?C?C?匹配CCC（D是可选的，它被忽略了）。MMMCCC在罗马数字中表示3300。
5. ‘MCMC’匹配失败。第一个M被匹配，第二个和第三个M被忽略，然后CM匹配成功。紧接着$试图匹配字符串结束，但后面是C，匹配失败。C也不能被D?C?C?C?匹配到，因为

CM和它只能匹配其中一个，而CM已经匹配过了。
6. 有趣的是，空字符串仍然可以匹配成功。因为所有的M都是可选的，都可以被忽略。并且后面的D?C?C?C?也是这种情况。

哈哈，看看正则表达式如此快速的处理了这些令人厌恶的东西。你已经可以找到千位数和百位数了！后面的十位和个位的处理和千位、百位的处理是一样的。但我们可以看看怎么用另一种方式来写这个正则表达式。

使用语法 `{n,m}`

{1,4} 匹配1到4个前面的模式

在上一节中，你处理过同样的字符可以重复0到3次的情况。实际上，还有另一种正则表达式的书写方式可以表达同样的意思，而且这种表达方式更具有可读性。首先看看我们在前面例子中使用的方法。

```
>>> import re
>>> pattern = '^M?M?M?$'

<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'

<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'

<_sre.SRE_Match object at 0x008EE090>

>>>
```

1. 正则表达式匹配字符串开始，然后是第一个可选的字符M，但没有第二个和第三个M（没问题！因为他们是可选的），接着是字符串结尾。
2. 正则表达式匹配字符串开始，然后是第一个和第二个M，第三个被忽略（因为它是可选的），最后匹配字符串结尾。
3. 正则表达式匹配字符串开始，然后是三个M，接着是字符串结尾。
4. 正则表达式匹配字符串开始，然后是三个M，但匹配字符串结尾失败（因为后面还有个M）。因此，这次匹配返回None。

```

<_sre.SRE_Match object at 0x008EEB48>

<_sre.SRE_Match object at 0x008EE090>

<_sre.SRE_Match object at 0x008EEDA8>

>>>
```

1. 这个正则表达式的意思是“匹配字符串开始，然后是任意的0到3个M字符，再是字符串结尾”。0和3的位置可以写任意的数字。如果你想表示可以匹配的最小次数为1次，最多为3次M字符，可以写成M{1,3}。

2. 匹配字符串开始，然后匹配了1次M，这在0到3的范围内，接着是字符串结尾。
3. 匹配字符串开始，然后匹配了2次M，这在0到3的范围内，接着是字符串结尾。
4. 匹配字符串开始，然后匹配了3次M，这在0到3的范围内，接着是字符串结尾。
5. 匹配字符串开始，然后匹配了3次M，这在0到3的范围内，但无法匹配后面的字符串结尾。正则表达式在字符串结尾之前最多允许匹配3次M，但这里有4个。因此本次匹配返回None。

检查十位和个位

现在，我们继续解释正则表达式匹配罗马数字中的十位和个位。下面的例子是检查十位。

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'

<_sre.SRE_Match object at 0x008EEB48>

<_sre.SRE_Match object at 0x008EEB48>

<_sre.SRE_Match object at 0x008EEB48>

<_sre.SRE_Match object at 0x008EEB48>

>>>
```

1. 匹配字符串开始，然后是第一个可选的M，接着是CM，XL，以及字符串结尾。记住：（A|B|C）的意思是“只匹配A，B或者C中的一个”。你匹配了XL，因此XC和L?X?X?X?被忽略，紧接着将检查字符串结尾。MCMXL在罗马数字中表示1940。
2. 匹配字符串开始，然后是第一个可选的M，接着是CM。后面的L被L?X?X?X?匹配，这里忽略掉L后面所有的X。然后检查字符串结尾。MCML在罗马数字中表示1950。
3. 匹配字符串开始，然后是第一个可选的M，接着是CM，还有可选的L以及第一个X，跳过后面的第二个和第三个X。然后检查字符串结尾。MCMLX表示1960。
4. 匹配字符串开始，然后是第一个可选的M，接着是CM，还有可选的L以及所有的三个X。然后是字符串结尾。MCMLXXX表示1980。
5. 匹配字符串开始，然后是第一个可选的M，接着是CM，还有可选的L以及所有的三个X。但匹配字符串结尾失败。因为后面还有一个X。整个匹配失败，返回None。MCMLXXXX不是一个合法的罗马数字。

(A|B) 匹配A模式或者B模式中的一个

个位数的匹配是同样的模式，我会告诉你细节以及最终结果。

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

使用{n,m}的语法来替代上面的写法会是什么样子呢？下面的例子展示了这种新的语法。

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
<_sre.SRE_Match object at 0x008EEB48>
<_sre.SRE_Match object at 0x008EEB48>
<_sre.SRE_Match object at 0x008EEB48>
<_sre.SRE_Match object at 0x008EEB48>
```

1. ^匹配字符串开始，然后表达式M{0,3}可以匹配0到3个的M。这里只能匹配一个M，也是可以的。接着，D?C{0,3}可以匹配一个可选的D，以及0到3个可能的C。这里我们实际只有一个D可以匹配到，正则表达式中的C全部忽略。往后，L?X{0,3}只能匹配到一个可选的L，没有X。接着V?I{0,3}匹配到一个可选的V，没有字符I。最后$匹配字符串结束。MDLV表示1555。
2. ^匹配字符串开始，然后匹配到2个M，D?C{0,3}匹配到可选的D，以及1个可能的C。往后，L?X{0,3}匹配到可选的L和1个X。接着V?I{0,3}匹配可选的V以及1个可选的I字符。最后匹配字符串结束。MMDCLXVI表示2666。
3. ^匹配字符串开始，然后是3个M，D?C{0,3}匹配到可选的D，以及3个C。往后，L?X{0,3}匹配可选的L和3个X。接着V?I{0,3}匹配可选的V以及3个I。最后匹配字符串结束。MMMDCCCLXXXVIII表示3888。这是你不用扩展语法写出来的最长罗马数字。
4. 靠近一点，（我就像一个魔术师："靠近一点，孩子们。我要从帽子里拿出一只兔子。"）^匹配字符串开始，然后M可以不被匹配（因为是匹配0到3次），接着匹配D?C{0,3}，这里跳过了可选的D，并且也没有匹配到C，下面L?X{0,3}也一样，跳过了L，没有匹配X。V?I{0,3}也跳过了V，匹配了1个I。然后匹配字符串结尾。太让人惊奇了！

如果你一次性就理解了上面所有的例子，那你会做的比我还好！现在想象一下以前的做法，在一个大程序用条件判断和函数来处理现在正则表达式处理的内容，或者想象一下前面写的正则表达式。我们发现，那些做法一点也不漂亮。

现在我们来研究一下怎么让你的正则表达式更具有维护性，但表达的意思却是相同的。

松散正则表达式

到目前为止，你只是处理了一些小型的正则表达式。就像你所看到的，他们难以阅读，甚至你不能保证半年后，你还能理解这些东西，并指出他们是干什么的。所以你需要在正则表达式内部添加一些说明信息。

python允许你使用松散正字表达式来达到目的。松散正字表达式和普通紧凑的正则表达式有两点不同：

- 空白符被忽略。空格、制表符和回车在正则表达式中并不会匹配空格、制表符、回车。如果你想在正则表达式中匹配他们，可以在前面加一个\来转义。
- 注释信息被忽略。松散正字表达式中的注释和python代码中的一样，都是以#开头直到行尾。它可以在多行正则表达式中增加注释信息，这就避免了在python代码中的多行注

释。他们的工作方式是一样的。

下面是一个更加清楚的例子。我们再来看看把上面的紧凑正则表达式改写成松散正字表达式后的样子。

```
>>> pattern = '''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    '''

<_sre.SRE_Match object at 0x008EEB48>

<_sre.SRE_Match object at 0x008EEB48>

<_sre.SRE_Match object at 0x008EEB48>
```

1. 注意，如果要使用松散正则表达式，需要传递一个叫re.VERBOSE的参数。就像你看到的那样，正则表达式中有很多空白符，他们都被忽略掉了。还有一些注释信息，当然也被正则表达式忽略掉。当空白符和注释信息被忽略掉后，这个正则表达式和上面的是完全一样的，但是它有更高的可读性。
2. 匹配字符串开始，然后是1个M，接着是CM，还有一个L和三个X，后面是IX，最后匹配字符串结尾。
3. 匹配字符串开始，然后是3个M，接着是D和三个C，以及三个X，一个V，三个I，最后匹配字符串结尾。
4. 这个不能匹配成功。为什么呢？因为他没有re.VERBOSE标记。因此search()会把他们整个当成一个紧凑的正则表达式，包括里面的空白符。python不会自动检测一个正则表达式是否是松散正则表达式，而需要明确的指定。⁂

案例研究: 解析电话号码

\d 匹配所有0-9的数字. \D 匹配除了数字外的所有字符.

到目前为止，我们主要关注于整个表达式是否能匹配到，要么整个匹配，要么整个都不匹配。但正则表达式还有更加强大的功能。如果正则表达式成功匹配，你可以找到正则表达式中某一部分匹配到什么。

这个例子来自于我在真实世界中遇到的另一个问题。这个问题是：解析一个美国电话号码。客户想用自由的格式来输入电话号码（在单个输入框），这需要存储区域码，交换码以及后四码（美国的电话分为区域码、交换码和后四码）。我在网上搜索，发现了很多解决这个问题的正则表达式，但是它们都能不完全满足我的要求。

下面是我要接受的电话号码格式：

- `800-555-1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555-1212`
- `1-800-555-1212`
- `800-555-1212-1234`
- `800-555-1212x1234`
- `800-555-1212 ext. 1234`
- `work 1-(800) 555.1212 #1234`

样式够多的！在上面的例子中，我知道区域码是800，交换码是555，以及最后的后四码是1212。如果还有分机号，那就是1234。

我们来解决这个电话号码解析问题。下面的例子是第一步。

```
('800', '555', '1212')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

1. 我们通常从左到右的阅读正则表达式。首先是匹配字符串开始位置，然后是(\d{3})。\d{3}表示什么意思？\d表示任意的数字（0到9），{3}表示一定要匹配3个数字。这个是你前面看到的{n,m}表示方法。把他们放在圆括号中，表示必须匹配3个数字，并且把他们记做一个组。分组的概念我们后面会说到。然后匹配一个连字符，接着匹配另外的3个数字，他们也同样作为一个组。然后又是一个连字符，后面还要准确匹配4个数字，他们也作为一位分组。最后匹配字符串结尾。
2. 为了使用正则表达式匹配到的这些分组，需要对search()函数的返回值调用groups()方法。它会返回一个这个正则表达式中定义的所有分组结果组成的元组。在这里，我们定义了三个分组，一个三个数字，另一个是三个数字，以及一个四个数字
3. 这个正则表达式并不是最终答案。因为它还没有处理有分机号的情况。为了处理这种情况，必须要对这个正则表达式进行扩展。
4. 这是为什么你不能在产品代码中链式调用search()和groups()的原因。如果search()方法匹配不成功，也就是返回None，这就不是返回的一个正则表达式匹配对象。它没有groups()方法，所以调用None.groups()将会抛出一个异常。（当然，在你的代码中，这个异常很明显。在这里我说了我的一些经验。）

```
('800', '555', '1212', '1234')

>>>

>>>
```

1. 这个正则表达式和前面的一样。匹配了字符串开始位置，然后是一个三个数字的分组，

接着一个连字符，又是一个三个数字的分组，又是一个连字符，然后一个四个数字的分组。这三个分组匹配的内容都会被记忆下来。和上面不同的是，这里多匹配了一个连字符以及一个分组，这个分组里的内容是匹配一个或更多个数字。最后是字符串结尾。
2. 现在groups()方法返回有四个元素的元组。因为正则表达式现在定义了四个组。
3. 不幸的是，这个正则表达式仍然不是最终答案。因为它假设这些数字是有连字符分隔的。实际上还有用空格，逗号和点分隔的情况。这就需要用更加通用的解决方案来匹配这些不同的分隔符。
4. 噢，这个正则表达式不但不能做到你想要的，而且还不如上一个了！因为我们现在不能匹配没有分机号的电话号码。这绝对不是你想要的。如果有分机号，你希望取到，但如果没有，你同样也希望匹配到电话号码其他的部分。

下面的例子展示了正则表达式中怎么处理电话号码中各个部分之间使用了不同分隔符的情况。

```
('800', '555', '1212', '1234')
('800', '555', '1212', '1234')
>>>
>>>
```

1. 注意了！你匹配了字符串开始，然后是3个数字的分组，接着是\D+，这是什么？好吧，\D匹配除了数字以外的任意字符，+的意思是一个或多个。因此\D+匹配一个或一个以上的非数字字符。这就是你用来替换连字符的东西，它用来匹配不同的分隔符。
2. 用\D+替换-，意味着你可以匹配分隔符为空格的情况。
3. 当然，分隔符为连字符一样可以正确工作。
4. 不幸的是，这仍然不是最终答案。因为这里我们假设有分隔符的存在，如果是根本就没有空格或者是连字符呢?
5. 天啊，它仍然没有解决分机号的问题。现在你有两个问题没有解决，但是我们可以用相同的技术来解决他们。

下面的例子展示用正则表达式处理电话号码没有分隔符的情况。

```
('800', '555', '1212', '1234')
('800', '555', '1212', '1234')
('800', '555', '1212', '')
>>>
```

1. 这里和上面唯一不同的地方是，把所有的+换成了。号码之间的分隔符不再用*\D*+来匹配，而是使用*\D*。还记得+表示一个或更多吧？好，现在可以解析号码之间没有分隔符的情况了。

2. 你看，它真的可以工作。为什么呢？首先匹配字符串开始，然后是3个数字的分组（800），分组匹配的内容会被记忆下来。然后是0个非数字分隔字符，然后又是3个数字的分组（555），同样也会被记忆下来。后面是0个非数字字符，接着是4个数字的分组（1212），然后又是0个非数字字符，还有一个任意个数字的分机号（1234）。最后匹配字符串结尾。
3. 其他字符作为分隔符一样可以工作。这里点替代了之前的连字符，分机号的前面还可以是空格和x。
4. 最后我们解决了这个长久以来的问题：分机号是可选的。如果分机号不存在，groups()仍然可以返回一个4元素的元组，只是第四个元素为空字符串。
5. 我讨厌坏消息。这还没有结束。还有什么问题呢？在区域码前面还可能有其他字符。但正则表达式假设区域码在字符串的开头。没关系，你还可以使用0个或更多的非数字字符串来跳过区位码前面的字符。

下面的例子展示怎么处理电话号码前面还有其他字符的情况。

```
('800', '555', '1212', '1234')
('800', '555', '1212', '')
>>>
```

1. 现在除了在第一个分组之前要用\d*匹配0个或更多非数字字符外，这和前面的例子是相同的。注意你不会对这些非数字字符分组，因为他们不在圆括号内，也就是说不是一个组。如果发现有这些字符，这里只是跳过他们，然后开始对后面的区域码匹配、分组。
2. 即使区位码之前有圆括号，你也可以成功的解析电话号码了。（右边的圆括号已经处理，它被\D*匹配成一个非数字字符。）
3. 这只是一个全面的检查，来确认以前能正确工作的现在仍然可以正确工作。因为首字符是可选的，因此首先匹配字符串开始，0个非数字字符，然后是三个数字并分组，接着是一个非数字字符，后面是三个数字并且分组，然后又是一个非数字分隔符，又是一个4个数字且分组，还有0个非数字字符，以及0个数字并且分组。最后匹配字符串结尾。
4. 还有问题。为什么不能匹配这个电话号码？因为在区域码前面还有一个1，但你假设的是区位码前面的第一个字符是非数字字符（\d*）

我们回过头看看。到目前为止，所有的正则表达式都匹配了字符串开始位置。但现在在字符串的开头可能有一些你想忽略掉的不确定的字符。为了匹配到想要的数据，你需要跳过他们。我们来看看不明确匹配字符串开始的方法。

```
('800', '555', '1212', '1234')
('800', '555', '1212', '')
('800', '555', '1212', '1234')
```

1. 注意正则表达式没有^。不会再匹配字符串开始位置了。正则表达式不会匹配整个字符

串，而是试图找到一个字符串开始匹配的位置，然后从这个位置开始匹配。
2. 现在，你可以正确的解析出字符串开头有不需要的字符、数字或者其他分隔符的情况了。
3. 全面性检查，同样正常工作了。
4. 这里也仍然可以工作。

看看正则表达式失控有多快？快速回顾一下之前的例子。你能说出他们的区别吗？

你看到了最终的答案（这就是最终答案！如果你发现还有它不能正确处理的情况，我也不想知道了）。在你忘掉它之前，我们来把它改写成松散正则表达式吧。

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non-digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of digits
    $           # end of string
    ''', re.VERBOSE)

('800', '555', '1212', '1234')

('800', '555', '1212', '')
```

1. 除了这里是用多行表示的以外，它和上面最后的那个是完全一样的。它一样可以处理之前的相同的情况。
2. 最后我们的全面检查也通过。很好，你终于完成了。

小结

这只是正则表达式能完成的工作中的冰山一角。换句话说，尽管你可能很受打击，相信我，你已经不是什么都不知道了。

现在，你应该已经熟悉了下面的技巧：

- `^` 匹配字符串开始位置。
- `$` 匹配字符串结束位置。
- `\b` 匹配一个单词边界。
- `\d` 匹配一个数字。
- `\D` 匹配一个任意的非数字字符。
- `x?` 匹配可选的x字符。换句话说，就是0个或者1个x字符。
- `x*` 匹配0个或更多的x。
- `x+` 匹配1个或者更多x。
- `x{n,m}` 匹配n到m个x，至少n个，不能超过m个。
- `(a|b|c)` 匹配单独的任意一个a或者b或者c。

- `(x)` 这是一个组，它会记忆它匹配到的字符串。你可以用re.search返回的匹配对象的groups()函数来获取到匹配的值。

正则表达式非常强大，但它也并不是解决每一个问题的正确答案。你需要更多的了解来判断哪些情况适合使用正则表达式。某些时候它可以解决你的问题，某些时候它可能带来更多的问题。

Chapter 6 闭合 与 生成器

> " My spelling is Wobbly. It's good spelling but it Wobbles, and the letters get in the wrong places. " — Winnie-the-Pooh

深入

出于传递所有理解的原因，我一直对语言非常着迷。我指的不是编程语言。好吧，是编程语言，但同时也是自然语言。使用英语。英语是一种七拼八凑的语言，它从德语、法语、西班牙语和拉丁语（等等）语言中借用了大量词汇。事实上，“借用”是不恰当的词汇，“掠夺”更加符合。或者也许叫“同化“——就像博格人（译注：根据维基百科资料，Borg 是《星际旅行》虚构宇宙中的一个种族，该译法未经原作者映证）。是的，我喜欢这样。

```
我们就是博格人。你们的语言和词源特性将会被添加到我们自己的当中。抵抗是徒劳的。
```

在本章中，将开始学习复数名词。以及返回其它函数的函数、高级正则表达式和生成器。但首先，让我们聊聊如何生成复数名词。（如果还没有阅读《正则表达式》一章，现在也许是个好时机读一读。本章将假定您理解了正则表达式的基础，并迅速进入更高级的用法。）

如果在讲英语的国家长大，或在正规的学校学习过英语，您可能对下面的基本规则很熟悉 ：

- 如果某个单词以 S 、X 或 Z 结尾，添加 ES 。*Bass* 变成 *basses*， *fax* 变成 *faxes*，而 *waltz* 变成 *waltzes*。
- 如果某个单词以发音的 H 结尾，加 ES；如果以不发音的 H 结尾，只需加上 S 。什么是发音的 H ？指的是它和其它字母组合在一起发出能够听到的声音。因此 *coach* 变成 *coaches* 而 *rash* 变成 *rashes*，因为在说这两个单词的时候，能够听到 CH 和 SH 的发音。但是 *cheetah* 变成 *cheetahs*，因为 H 不发音。
- 如果某个单词以发 I 音的字母 Y 结尾，将 Y 改成 IES；如果 Y 与某个原因字母组合发其它音的话，只需加上 S 。因此 *vacancy* 变成 *vacancies*，但 *day* 变成 *days* 。
- 如果所有这些规则都不适用，只需加上 S 并作最好的打算。

（我知道，还有许多例外情况。*Man* 变成 *men* 而 *woman* 变成 *women*，但是 *human* 变成 *humans*。*Mouse* 变成 *mice* ； *louse* 变成 *lice*，但 *house* 变成 *houses*。*Knife* 变成 *knives* ；*wife* 变成 *wives*，但是 *lowlife* 变成 *lowlifes*。而且甚至我还没有开始提到那些原型和复数形式相同的单词，就像 *sheep*、 *deer* 和 *haiku*。）

其它语言，当然是完全不同的。

让我们设计一个 Python 类库用来自动进行英语名词的复数形式转换。我们将以这四条规则为起点，但要记住的不可避免地还要增加更多规则。

我知道，让我们用正则表达式！

因此，您正在看着单词，至少是英语单词，也就是说您正在看着字符的字符串。规则说你必须找到不同的字符组合，然后进行不同的处理。这听起来是正则表达式的工作！

[下载 `plural1.py`]

```
import re

def plural(noun):

    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

1. 这是一条正则表达式，但它使用了在《正则表达式》一章中没有讲过的语法。中括号表示“匹配这些字符的其中之一”。因此 `[sxz]` 的意思是：“ `s` 、 `x` 或 `z` ”，但只匹配其中之一。对 `$` 应该很熟悉了，它匹配字符串的结尾。经过组合，该正则表达式将测试 `noun` 是否以 `s` 、 `x` 或 `z` 结尾。
2. 该 `re.sub()` 函数执行基于正则表达式的字符串替换。

让我们看看正则表达式替换的细节。

```
>>> import re
<_sre.SRE_Match object at 0x001C1FA8>
'Mork'
'rook'
'oops'
```

1. 字符串 `Mark` 包含 `a` 、 `b` 或 `c` 吗？是的，它包含 `a` 。
2. 好了，现在查找 `a` 、 `b` 或 `c`，并将其替换为 `o` 。 `Mark` 变成了 `Mork` 。
3. 同一函数将 `rock` 转换为 `rook` 。
4. 您可能会认为该函数会将 `caps` 转换为 `oaps`，但实际上并是这样。 `re.sub` 替换 所有的 匹配项，而不仅仅是第一个匹配项。因此该正则表达式将 `caps` 转换为 `oops`，因为无论是 `c` 还是 `a` 均被转换为 `o` 。

接下来，回到 `plural()` 函数......

```
def plural(noun):
    if re.search('[sxz]$', noun):

        return re.sub('$', 'es', noun)

        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

1. 此处将字符串的结尾（通过 `$` 匹配）替换为字符串 `es` 。换句话来说，向字符串尾部添加一个 `es` 。可以通过字符串链接来完成同样的变化，例如 `noun + 'es'` ，但我对每条规则都选用正则表达式，其原因将在本章稍后更加清晰。
2. 仔细看看，这里出现了新的变化。作为方括号中的第一个字符， `^` 有特别的含义：非。 `[^abc]` 的意思是：“ 除了 `a` 、 `b` 或 `c` 之外的任何字符”。因此 `[^aeioudgkprt]` 的意思是除了 `a` 、 `e` 、 `i` 、 `o` 、 `u` 、 `d` 、 `g` 、 `k` 、 `p` 、 `r` 或 `t` 之外的任何字符。然后该字符必须紧随一个 `h` ，其后是字符串的结尾。所匹配的是以 H 结尾且 H 发音的单词。
3. 此处有同样的模式：匹配以 Y 结尾的单词，而 Y 之前的字符 不是 `a` 、 `e` 、 `i` 、 `o` 或 `u` 。所匹配的是以 Y 结尾，且 Y 发音听起来像 I 的单词。

让我们看看“否定”正则表达式的更多细节。

```
>>> import re

<_sre.SRE_Match object at 0x001C1FA8>

>>>
>>> re.search('[^aeiou]y$', 'day')
>>>

>>>
```

1. `vacancy` 匹配该正则表达式，因为它以 `cy` 结尾，且 `c` 并非 `a` 、 `e` 、 `i` 、 `o` 或 `u` 。
2. `boy` 不匹配，因为它以 `oy` 结尾，可以明确地说 `y` 之前的字符不能是 `o` 。 `day` 不匹配，因为它以 `ay` 结尾。
3. `pita` 不匹配，因为它不以 `y` 结尾。

```

'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'

'vacancies'
```

1. 该正则表达式将 `vacancy` 转换为 `vacancies` ，将 `agency` 转换为 `agencies` ，这正是想要的结果。注意，它也会将 `boy` 转换为 `boies` ，但这永远也不会在函数中发生，因为我们首先进行了 `re.search` 以找出永远不应进行该 `re.sub` 操作的单词。
2. 顺便，我还想指出可以将该两条正则表达式合并起来（一条查找是否应用该规则，另一

条实际应用规则），使其成为一条正则表达式。它看起来是下面这个样子：其中多数内容看起来应该很熟悉：使用了在 案例研究：分析电话号码 中用到的记忆分组。该分组用于保存字母 `y` 之前的字符。然后在替换字符串中，用到了新的语法： `\1` ，它表示“嘿，记住的第一个分组呢？把它放到这里。”在此例中， 记住了 `y` 之前的 `c` ，在进行替换时，将用 `c` 替代 `c` ，用 `ies` 替代 `y` 。（如果有超过一个的记忆分组，可以使用 `\2` 和 `\3` 等等。）

正则表达式替换功能非常强大，而 `\1` 语法则使之愈加强大。但是，将整个操作组合成一条正则表达式也更难阅读，而且也没有直接映射到刚才所描述的复数规则。刚才所阐述的规则，像 “如果单词以 S 、X 或 Z 结尾，则添加 ES 。”如果查看该函数，有两行代码都在表述“如果以 S 、X 或 Z 结尾，那么添加 ES 。”它没有之前那种模式更直接。

函数列表

现在要增加一些抽象层次的内容。我们开始时定义了一系列规则：如果这样，那样做；否则前往下一条规则。现在让我们对部分程序进行临时的复杂化，以简化另一部分。

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

    return re.search('[^aeiou]y$', noun)

    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return True

def apply_default(noun):
    return noun + 's'

         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )

def plural(noun):

        if matches_rule(noun):
            return apply_rule(noun)
```

1. 现在，每条匹配规则都有自己的函数，它们返回对 `re.search()` 函数调用结果。
2. 每条应用规则也都有自己的函数，它们调用 `re.sub()` 函数以应用恰当的复数变化规则。

3. 现在有了一个 `rules` 数据结构——一个函数对的序列，而不是一个函数（ `plural()` ）实现多个条规则。
4. 由于所有的规则被分割成单独的数据结构，新的 `plural()` 函数可以减少到几行代码。使用 `for` 循环，可以一次性从 `rules` 这个数据结构中取出匹配规则和应用规则这两样东西（一条匹配对应一条应用）。在 `for` 循环的第一次迭代过程中， `matches_rule` 将获取 `match_sxz` ，而 `apply_rule` 将获取 `apply_sxz` 。在第二次迭代中（假定可以进行到这一步）， `matches_rule` 将会赋值为 `match_h` ，而 `apply_rule` 将会赋值为 `apply_h` 。该函数确保最终能够返回某个值，因为终极匹配规则 (`match_default`) 只返回 `True` ，意思是对应的应用规则 (`apply_default`) 将总是被应用。

变量 “rules” 是一系列函数对。

该技术能够成功运作的原因是 Python 中一切都是对象，包括了函数。数据结构 `rules` 包含了函数——不是函数的名称，而是实际的函数对象。在 `for` 循环中被赋值后， `matches_rule` 和 `apply_rule` 是可实际调用的函数。在第一次 `for` 循环的迭代过程中，这相当于调用 `matches_sxz(noun)` ，如果返回一个匹配值，将调用 `apply_sxz(noun)` 。

如果这种附加抽象层令你迷惑，可以试着展开函数以了解其等价形式。整个 `for` 循环等价于下列代码：

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

这段代码的好处是 `plural()` 函数被简化了。它处理一系列其它地方定义的规则，并以通用的方式对它们进行迭代。

1. 获取某匹配规则
2. 是否匹配？然后调用应用规则，并返回结果。
3. 不匹配？返回步骤 1 。

这些规则可在任何地方以任何方式定义。 `plural()` 函数并不关心。

现在，新增的抽象层是否值得呢？嗯，还没有。让我们考虑下要向函数中新增一条规则时该如何操作。在第一例中，将需要新增一条 `if` 语句到 `plural()` 函数中。在第二例中，将需要新增两个函数， `match_foo()` 和 `apply_foo()` ，然后更新 `rules` 序列以指定新的匹配和应用函数按照其它规则按顺序调用。

但是对于下一节来说，这只是一个跳板而已。让我们继续……

匹配模式列表

其实并不是真的有必要为每个匹配和应用规则定义各自的命名函数。它们从未直接被调用，而只是被添加到 `rules` 序列并从该处被调用。此外，每个函数遵循两种模式的其中之一。所有的匹配函数调用 `re.search()`，而所有的应用函数调用 `re.sub()`。让我们将模式排除在考虑因素之外，使新规则定义更加简单。

```
import re

def build_match_and_apply_functions(pattern, search, replace):

        return re.search(pattern, word)

        return re.sub(search, replace, word)
```

1. `build_match_and_apply_functions()` 函数用于动态创建其它函数。它接受 `pattern`、`search` 和 `replace` 三个参数，并定义了 `matches_rule()` 函数，该函数通过传给 `build_match_and_apply_functions()` 函数的 `pattern` 及传递给所创建的 `matchs_rules()` 函数的 `word` 调用 `re.search()` 函数，哇。
2. 应用函数的创建工作采用了同样的方式。应用函数只接受一个参数，并使用传递给 `build_match_and_apply_functions()` 函数的 `search` 和 `replace` 参数、以及传递给要创建 `apply_rule()` 函数的 `word` 调用 `re.sub()`。在动态函数中使用外部参数值的技术称为 闭合【*closures*】。基本上，常量的创建工作都在创建应用函数过程中完成：它接受一个参数 （ `word` ），但实际操作还加上了另外两个值（ `search` 和 `replace` ），该两个值都在定义应用函数时进行设置。
3. 最后， `build_match_and_apply_functions()` 函数返回一个包含两个值的元组：即刚才所创建的两个函数。在这些函数中定义的常量（ `match_rule()` 函数中的 `pattern` 函数， `apply_rule()` 函数中的 `search` 和 `replace` ）与这些函数呆在一起，即便是在从 `build_match_and_apply_functions()` 中返回后也一样。这真是非常酷的一件事情。

但如果此方式导致了难以置信的混乱（应该是这样，它确实有点奇怪），在看看如何使用之后可能会清晰一些。

```
  (
    ('[sxz]$',           '$',  'es'),
    ('[^aeioudgkprt]h$', '$',  'es'),
    ('(qu|[^aeiou])y$',  'y$', 'ies'),

  )

         for (pattern, search, replace) in patterns]
```

1. 我们的复数形式“规则”现在被定义为 字符串 的元组的元组（而不是函数）。每个组的第一个字符串是在 `re.search()` 中用于判断该规则是否匹配的正则表达式。各组中的第二和第三个字符串是在 `re.sub()` 中将实际用于使用规则将名词转换为复数形式的搜索和替换表达式。
2. 此处的后备规则略有变化。在前例中， `match_default()` 函数仅返回 `True`，意思是如果更多的指定规则无一匹配，代码将简单地向给定词汇的尾部添加一个 `s` 。本例则进行了

一些功能等同的操作。最后的正则表达式询问单词是否有一个结尾（ `$` 匹配字符串的结尾）。当然，每个字符串都有一个结尾，甚至是空字符串也有，因此该规则将始终被匹配。因此，它实现了 `match_default()` 函数同样的目的，始终返回 `True` ：它确保了如果没有更多的指定规则用于匹配，代码将向给定单词的尾部增加一个 `s` 。

3. 本行代码非常神奇。它以 `patterns` 中的字符串序列为参数，并将其转换为一个函数序列。怎么做到的？通过将字符串"映射"到 `build_match_and_apply_functions()` 函数。也就是说，它接受每组三重字符串为参数，并将该三个字符串作为实参调用 `build_match_and_apply_functions()` 函数。 `build_match_and_apply_functions()` 函数返回一个包含两个函数的元组。也就是说该 `规则` 最后的结尾与前例在功能上是等价的：一个元组列表，每个元组都是一对函数。第一个函数是调用 `re.search()` 的匹配函数；而第二个函数调用 `re.sub()` 的应用函数。

此版本脚本的最前面是主入口点—— `plural()` 函数。

```
def plural(noun):

        if matches_rule(noun):
            return apply_rule(noun)
```

1. 由于 `规则` 列表与前例中的一样（实际上确实相同），因此毫不奇怪 `plural()` 函数基本没有发生变化。它是完全通用的，它以规则函数列表为参数，并按照顺序调用它们。它并不关系规则是如何定义的。在前例中，它们被定义为各自命名的函数。现在它们通过将 `build_match_and_apply_functions()` 函数的输出映射为源字符串的列表来动态创建。这没有任何关系； `plural()` 函数将以同样方式运作。

匹配模式文件

目前，已经排除了重复代码，增加了足够的抽象性，因此复数形式规则可以字符串列表的形式进行定义。下一个逻辑步骤是将这些字符串放入一个单独的文件中，因此可独立于使用它们的代码来进行维护。

首先，让我们创建一份包含所需规则的文本文件。没有花哨的数据结构，只有空白符分隔的三列字符串。将其命名为 `plural4-rules.txt` .

```
[sxz]$              $    es
[^aeioudgkprt]h$    $    es
[^aeiou]y$         y$    ies
$                   $    s
```

下面看看如何使用该规则文件。

```
import re

    def matches_rule(word):
        return re.search(pattern, word)
    def apply_rule(word):
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)

rules = []

                pattern, search, replace))
```

1. `build_match_and_apply_functions()` 函数没有发生变化。仍然使用了闭合技术：通过外部函数中定义的变量来动态创建两个函数。
2. 全局的 `open()` 函数打开文件并返回一个文件对象。此例中，将要打开的文件包含了名词复数形式的模式字符串。 `with` 语句创建了叫做 *context*【上下文】的东西：当 `with` 块结束时，Python 将自动关闭文件，即便是在 `with` 块中引发了例外也会这样。在 《文件》 一章中将学到关于 `with` 块和文件对象的更多内容。
3. `for line in <fileobject>` 代码从打开的文件中读取数据，并将文本赋值给 `line` 变量。在 《文件》 一章中将学到更多关于读取文件的内容。
4. 文件中每行都有三个值，单它们通过空白分隔（制表符或空白，没有区别）。要将它们分开，可使用字符串方法 `split()` 。 `split()` 方法的第一个参数是 `None` ，表示"对任何空白字符进行分隔（制表符或空白，没有区别）"。第二个参数是 `3` ，意思是"针对空白分隔三次，丢弃该行剩下的部分。"像 `[sxz]$ $ es` 这样的行将被分割为列表 `['[sxz]$', '$', 'es']` ，意思是 `pattern` 获得值 `'[sxz]$'` ， `search` 获得值 `'$'` ，而 `replace` 获得值 `'es'` 。对于短短的一行代码来说确实威力够大的。
5. 最后，将 `pattern` 、 `search` 和 `replace` 传入 `build_match_and_apply_functions()` 函数，它将返回一个函数的元组。将该元组添加到 `rules` 列表，最终 `rules` 将储存 `plural()` 函数所预期的匹配和应用函数列表。

此处的改进是将复数形式规则独立地放到了一份外部文件中，因此可独立于使用它的代码单独对规则进行维护。代码是代码，数据是数据，生活更美好。

生成器

如果有个通用 `plural()` 函数解析规则文件不就更棒了吗？获取规则，检查匹配，应用相应的转换，进入下一条规则。这是 `plural()` 函数所必须完成的事，也是 `plural()` 函数必须做的事。

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)
            yield build_match_and_apply_functions(pattern, search, replace)

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

这段代码到底是如何运作的？让我们先看一个交互式例子。

```
>>> def make_counter(x):
...     print('entering make_counter')
...     while True:

...         print('incrementing x')
...         x = x + 1
...

<generator object at 0x001C9C10>

entering make_counter
2

incrementing x
3

incrementing x
4
```

1. `make_counter` 中出现的 `yield` 命令的意思是这不是一个普通的函数。它是一次生成一个值的特殊类型函数。可以将其视为可恢复函数。调用该函数将返回一个可用于生成连续 `x` 值的 生成器【*Generator*】。
2. 为创建 `make_counter` 生成器的实例，仅需像调用其它函数那样对它进行调用。注意该调用并不实际执行函数代码。可以这么说，是因为 `make_counter()` 函数的第一行调用了 `print()` ，但实际并未打印任何内容。
3. 该 `make_counter()` 函数返回了一个生成器对象。
4. `next()` 函数以一个生成器对象为参数，并返回其下一个值。对 `counter` 生成器第一次调用 `next()` ，它针对第一条 `yield` 语句执行 `make_counter()` 中的代码，然后返回所产生的值。在此情况下，该代码输出将为 `2` ，因其仅通过调用 `make_counter(2)` 对生成器进行初始创建。
5. 对同一生成器对象反复调用 `next()` 将确切地从上次调用的位置开始继续，直到下一条 `yield` 语句。所有的变量、局部数据等内容在 `yield` 时被保存，在 `next()` 时被恢复。下一行代码等待被执行以调用 `print()` 以打印出 `incrementing x` 。之后，执行语句 `x = x + 1` 。然后它继续通过 `while` 再次循环，而它再次遇上的第一条语句是 `yield x` ，该语句将保存所有一切状态，并返回当前 `x` 的值（当前为 `3` ）。
6. 第二次调用 `next(counter)` 时，又进行了同样的工作，但这次 `x` 为 `4` 。

由于 `make_counter` 设置了一个无限循环，理论上可以永远执行该过程，它将不断递增 `x` 并输出数值。还是让我们看一个更加实用的生成器用法。

斐波那奇生成器

“yield” 暂停一个函数。“next()” 从其暂停处恢复其运行。

```
def fib(max):
    while a < max:
```

1. 斐波那契序列是一系列的数字，每个数字都是其前两个数字之和。它从 0 和 `1` 开始，初始时上升缓慢，但越来越快。启动该序列需要两个变量：从 0 开始的 `a`，和从 `1` 开始的 `b` 。
2. `a` 是当前序列中的数字，因此对它进行 yield 操作。
3. `b` 是序列中下一个数字，因此将它赋值给 `a`，但同时计算下一个值 (`a + b`) 并将其赋值给 `b` 以供稍后使用。注意该步骤是并行发生的；如果 `a` 为 `3` 且 `b` 为 `5`，那么 `a, b = b, a + b` 将会把 `a` 设置 `5` （ `b` 之前的值），将 `b` 设置为 `8` （ `a` 和 `b` 之前值的和）。

因此，现在有了一个连续输出斐波那契数值的函数。当然，还可以使用递归来完成该功能，但这个方式更易于阅读。同样，它也与 `for` 循环合作良好。

```
>>> from fibonacci import fib
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

1. 可以在 `for` 循环中直接使用像 `fib()` 这样的生成器。 `for` 循环将会自动调用 `next()` 函数，从 `fib()` 生成器获取数值并赋值给 `for` 循环索引变量。（ `n` ）
2. 每经过一次 `for` 循环， `n` 从 `fib()` 的 `yield` 语句获取一个新值，所需做的仅仅是输出它。一旦 `fib()` 的数字用尽（ `a` 大于 `max`，即本例中的 `1000` ）， `for` 循环将会自动退出。
3. 这是一个很有用的用法：将一个生成器传递给 `list()` 函数，它将遍历整个生成器（就像前例中的 `for` 循环）并返回所有数值的列表。

复数规则生成器

让我们回到 `plural5.py` 看看该版本的 `plural()` 函数是如何运作的。

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:

def plural(noun, rules_filename='plural5-rules.txt'):

        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

1. 此处没有太神奇的代码。由于规则文件中每行都靠包括以空白相间的三个值，因此使用 `line.split(None, 3)` 获取三个“列”的值并将它们赋值给三个局部变量。
2. 然后使用了 *yield*。 但生产了什么呢？通过老朋友—— `build_match_and_apply_functions()` 动态创建的两个函数，这与之前的例子是一样的。换而言之， `rules()` 是按照需求连续生成匹配和应用函数的生成器。
3. 由于 `rules()` 是生成器，可直接在 `for` 循环中使用它。对 `for` 循环的第一次遍历，可以调用 `rules()` 函数打开模式文件，读取第一行，从该行的模式动态创建一个匹配函数和应用函数，然后生成动态创建的函数。对 `for` 循环的第二次遍历，将会精确地回到 `rules()` 中上次离开的位置（在 `for line in pattern_file` 循环的中间）。要进行的第一项工作是读取文件（仍处于打开状态）的下一行，基于该行的模式动态创建另一匹配和应用函数，然后生成两个函数。

通过第四步获得了什么呢？启动时间。在第四步中引入 `plural4` 模块时，它读取了整个模式文件，并创建了一份所有可能规则的列表，甚至在考虑调用 `plural()` 函数之前。有了生成器，可以轻松地处理所有工作：可以读取规则，创建函数并试用它们，如果该规则可用甚至可以不读取文件剩下的部分或创建更多的函数。

失去了什么？性能！每次调用 `plural()` 函数， `rules()` 生成器将从头开始——这意味着重新打开模式文件，并从头开始读取，每次一行。

要是能够两全其美多好啊：最低的启动成本（无需对 `import` 执行任何代码），同时 最佳的性能（无需一次次地创建同一函数）。哦，还需将规则保存在单独的文件中（因为代码和数据要泾渭分明），还有就是永远不必两次读取同一行。

要实现该目标，必须建立自己的生成器。在进行此工作之前，必须对 Python 的类进行学习。

深入阅读

- PEP 255: 简单生成器
- 理解 Python 的 “with” 语句
- Python 中的闭合
- 斐波那契数值
- 英语的不规则复数名词

Chapter 7 类 & 迭代器

> " 东是东，西是西，东西不相及 " — 拉迪亚德·吉卜林

深入

生成器是一类特殊 迭代器。 一个产生值的函数 `yield` 是一种产生一个迭代器却不需要构建迭代器的精密小巧的方法。 我会告诉你我是什么意思。

记得 菲波拉稀生成器吗？ 这里是一个从无到有的迭代器：

[下载 `fibonacci2.py`]

```
class Fib:
    '''生成菲波拉稀数列的迭代器'''

    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

让我们一行一行来分析。

```
class Fib:
```

类(class)？什么是类？

类的定义

Python 是完全面向对象的：你可以定义自己的类，从你自己或系统自带的类继承，并生成实例。

在Python里定义一个类非常简单。就像函数一样， 没有分开的接口定义。 只需定义类就开始编码。 Python类以保留字 `class` 开始， 后面跟类名。 技术上来说，只需要这么多就够了，因为一个类不是必须继承其他类。

1. 类名是 `PapayaWhip`， 没有从其他类继承。 类名通常是大写字母分隔，

如 `EachWordLikeThis` ， 但这只是个习惯，并非必须。
2. 你可能猜到，类内部的内容都需缩进，就像函数中的代码一样， `if` 语句， `for` 循环， 或其他代码块。第一行非缩进代码表示到了类外。

`PapayaWhip` 类没有定义任何方法和属性， 但依据句法，应该在定义中有东西，这就是 `pass` 语句。 这是Python 保留字，意思是“继续，这里看不到任何东西”。 这是一个什么都不做的语句，是一个很好的占位符，如果你的函数和类什么都不想做（删空函数或类）。

> ☞Python中的 `pass` 就像Java 或 C中的空大括号对 (`{}`) 。

很多类继承自其他类， 但这个类没有。 很多类有方法，这个类也没有。 Python 类不是必须有东西，除了一个名字。 特别是C++ 程序员发现 Python 类没有显式的构造和析构函数会觉得很古怪。 尽管不是必须， Python 类 可以 具有类似构造函数的东西： `__init__()` 方法。

`__init__()` 方法

本示例展示 `Fib` 类使用 `__init__` 方法。

```
class Fib:
```

1. 类同样可以 (而且应该) 具有 `docstring` ， 与模块和方法一样。
2. 类实例创建后， `__init__()` 方法被立即调用。很容易将其——但技术上来说不正确——称为该类的“构造函数” 。 很容易，因为它看起来很像 C++ 的构造函数（按约定， `__init__()` 是类中第一个被定义的方法），行为一致（是类的新实例中第一片被执行的代码）， 看起来完全一样。 错了， 因为 `__init__()` 方法调用时，对象已经创建了，你已经有了一个合法类对象的引用。

每个方法的第一个参数，包括 `__init__()` 方法，永远指向当前的类对象。 习惯上，该参数叫 `self` 。 该参数和C++或Java中 `this` 角色一样， 但 `self` 不是 Python的保留字， 仅仅是个命名习惯。 虽然如此，请不要取别的名字，只用 `self` ；这是一个很强的命名习惯。

在 `__init__()` 方法中， `self` 指向新创建的对象； 在其他类对象中， 它指向方法所属的实例。尽管需在定义方法时显式指定 `self` ，调用方法时并 不 必须明确指定。 Python 会自动添加。

实例化类

Python 中实例化类很直接。 实例化类时就像调用函数一样简单，将 `__init__()` 方法需要的参数传入。 返回值就是新创建的对象。

```
>>> import fibonacci2
<fibonacci2.Fib object at 0x00DB8810>
<class 'fibonacci2.Fib'>
''
```

1. 你正创建一个 `Fib` 类的实例（在 `fibonacci2` 模块中定义） 将新创建的实例赋给变量 `fib` 。 你传入一个参数 `100` ， 这是 `Fib` 的 `__init__()` 方法作为 `max` 参数传入的结束值。
2. `fib` 是 `Fib` 的实例。
3. 每个类实例具有一个内建属性， `__class__` ， 它是该对象的类。 Java 程序员可能熟悉 `Class` 类， 包含方法如 `getName()` 和 `getSuperclass()` 获取对象相关元数据。 Python 里面， 这类元数据由属性提供，但思想一致。
4. 你可访问对象的 `docstring` ，就像函数或模块中的一样。 类的所有实例共享一份 `docstring` 。

> ☞Python里面， 和调用函数一样简单的调用一个类来创建该类的新实例。 与C++ 或 Java不一样，没有显式的 `new` 操作符。

实例变量

继续下一行：

```
class Fib:
    def __init__(self, max):
```

1. `self.max` 是什么？ 它就是实例变量。 与作为参数传入 `__init__()` 方法的 `max` 完全是两回事。 `self.max` 是实例内 “全局” 的。 这意味着可以在其他方法中访问它。

```
class Fib:
    def __init__(self, max):

    .
    .
    .
    def __next__(self):
        fib = self.a
```

1. `self.max` 在 `__init__()` 方法中定义……
2. ……在 `__next__()` 方法中引用。

实例变量特定于某个类的实例。 例如， 如果你创建 `Fib` 的两个具有不同最大值的实例， 每个实例会记住自己的值。

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```

菲波拉稀迭代器

现在 你已经准备学习如何创建一个迭代器了。 迭代器就是一个定义了 `__iter__()` 方法的类。

这些类的所有三种方法， `__init__` ， `__iter__` ， 和 `__next__` ， 起始和结束均为一对下划线（ `_` ） 字符。 为什么这样？ 并无什么神奇之处， 只是通常表示这是“特殊方法。” 唯一“特殊”的地方，就是这些方法不是直接调用的； 当你使用类或实例的某些语法时，Python会自动调用他们。 更多关于特殊方法。

[下载 `fibonacci2.py`]

```
        self.max = max

        self.a = 0
        self.b = 1
        return self

        fib = self.a
        if fib > self.max:

        self.a, self.b = self.b, self.a + self.b
```

1. 从无到有创建一个迭代器， `fib` 应是一个类，而不是一个函数。
2. “调用” `Fib(max)` 会创建该类一个真实的实例，并以 `max` 做为参数调用 `__init__()` 方法。 `__init__()` 方法以实例变量保存最大值，以便随后的其他方法可以引用。
3. 当有人调用 `iter(fib)` 的时候， `__iter__()` 就会被调用。（正如你等下会看到的， `for` 循环会自动调用它， 你也可以自己手动调用。） 在完成迭代器初始化后，（在本例中，重置我们两个计数器 `self.a` 和 `self.b` ）， `__iter__()` 方法能返回任何实现了 `__next__()` 方法的对象。 在本例（甚至大多数例子）中， `__iter__()` 仅简单返回 `self` ， 因为该类实现了自己的 `__next__()` 方法。
4. 当有人在迭代器的实例中调用 `next()` 方法时， `__next__()` 会自动调用。 随后会有更多理解。
5. 当 `__next__()` 方法抛出 `StopIteration` 异常， 这是给调用者表示迭代用完了的信号。 和大多数异常不同， 这不是错误；它是正常情况，仅表示迭代器没有值可产生了。 如果调用者是 `for` 循环， 它会注意到该 `StopIteration` 异常并优雅的退出。 （换句话说，它会吞掉该异常。） 这点神奇之处就是使用 `for` 的关键。
6. 为了分离出下一个值， 迭代器的 `__next__()` 方法简单 `return` 该值。 不要使用 `yield`

；该语法上的小甜头仅用于你使用生成器的时候。 这里你从无到有创建迭代器，使用 `return` 代替。

完全晕了？ 太好了。 让我们看如何调用该迭代器：

```
>>> from fibonacci2 import Fib
>>> for n in Fib(1000):
...     print(n, end=' ')
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

为什么？完全一模一样！ 一字节一字节的与你调用 Fibonacci-as-a-generator （模块第一个字母大写）相同。但怎么做到的？

`for` 循环内有魔力。下面是究竟发生了什么：

- 如你所见， `for` 循环调用 `Fib(1000)` 。 这返回 `Fib` 类的实例。 叫它 `fib_inst` 。
- 背地里，且十分聪明的， `for` 循环调用 `iter(fib_inst)` ， 它返回迭代器。 叫它 `fib_iter` 。 本例中， `fib_iter` == `fib_inst` ， 因为 `__iter__()` 方法返回 `self` ，但 `for` 循环不知道（也不关心）那些。
- 为“循环通过”迭代器， `for` 循环调用 `next(fib_iter)` ， 它又调用 `fib_iter` 对象的 `__next__()` 方法， 产生下一个菲波拉稀计算并返回值。 `for` 拿到该值并赋给 `n` ， 然后执行 `n` 值的 `for` 循环体。
- `for` 循环如何知道什么时候结束？很高兴你问到。 当 `next(fib_iter)` 抛出 `StopIteration` 异常时， `for` 循环将吞下该异常并优雅退出。 （其他异常将传过并如常抛出。） 在哪里你见过 `StopIteration` 异常？ 当然在 `__next__()` 方法。

复数规则迭代器

iter(f) 调用 f.**iter** next(f) 调用 f.**next**

现在到曲终的时候了。我们重写 复数规则生成器 为迭代器。

[下载 `plural6.py`]

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')
        self.cache = []

    def __iter__(self):
        self.cache_index = 0
        return self

    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]

        if self.pattern_file.closed:
            raise StopIteration

        line = self.pattern_file.readline()
        if not line:
            self.pattern_file.close()
            raise StopIteration

        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(
            pattern, search, replace)
        self.cache.append(funcs)
        return funcs

rules = LazyRules()
```

因此这是一个实现了 `__iter__()` 和 `__next__()` 的类。所以它可以 被用作迭代器。然后，你实例化它并将其赋给 `rules` 。这只发生一次，在import的时候。

让我们一口一口来吃：

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
```

1. 当我们实例化 `LazyRules` 类时， 打开模式文件，但不读取任何东西。 （随后再进行）
2. 打开模式文件之后，初始化缓存。 随后读取模式文件行的时候会用到它（在 `__next__()` 方法中） 。

我们继续之前，让我们近观 `rules_filename` 。它没在 `__iter__()` 方法中定义。事实上，它没在任何方法中定义。它定义于类级别。它是 类变量， 尽管访问时和实例变量一样（ `self.rules_filename` ）， `LazyRules` 类的所有实例共享该变量。

```
>>> import plural6
>>> r1 = plural6.LazyRules()
>>> r2 = plural6.LazyRules()

'plural6-rules.txt'
>>> r2.rules_filename
'plural6-rules.txt'

>>> r2.rules_filename
'r2-override.txt'
>>> r1.rules_filename
'plural6-rules.txt'

'plural6-rules.txt'

>>> r1.rules_filename
'papayawhip.txt'

'r2-overridetxt'
```

1. 类的每个实例继承了 `rules_filename` 属性及它在类中定义的值。
2. 修改一个实例属性的值不影响其他实例……
3. ……也不会修改类的属性。可以使用特殊的 `__class__` 属性来访问类属性（于此相对的是单独实例的属性）。
4. 如果修改类属性， 所有仍然继承该实例的值的实例 （如这里的 `r1` ） 会受影响。
5. 已经覆盖（overridden）了该属性（如这里的 `r2` ）的所有实例 将不受影响。

现在回到我们的演示：

```
self.cache_index = 0
```

1. 无论何时有人——如 `for` 循环——调用 `iter(rules)` 的时候， `__iter__()` 方法都会被调用。
2. 每个 `__iter__()` 方法都需要做的就是必须返回一个迭代器。 在本例中，返回 `self` ，意味着该类定义了 `__next__()` 方法，由它来关注整个迭代过程中的返回值。

```
        .
        .
        .
        pattern, search, replace = line.split(None, 3)

            pattern, search, replace)

        return funcs
```

1. 无论何时有人——如 `for` 循环——调用 `__next__()` 方法， `next(rules)` 都跟着被调用。 该方法仅在我们从后往前移动时比较好体会。所以我们就这么做。
2. 函数的最后一部分至少应该眼熟。 `build_match_and_apply_functions()` 函数还没修改；与它从前一样。
3. 唯一的不同是，在返回匹配和应用功能之前（保存在元组 `funcs` 中），我们将其保存到 `self.cache` 。

从后往前移动……

```
def __next__(self):
        .
        .
        .
            self.pattern_file.close()

        .
        .
        .
```

1. 这里有点高级文件操作的技巧。 `readline()` 方法 （注意：是单数，不是复数 `readlines()`） 从一个打开的文件中精确读取一行，即下一行。（文件对象同样也是迭代器！ 它自始至终是迭代器……）
2. 如果有一行 `readline()` 可以读， `line` 就不会是空字符串。 甚至文件包含一个空行， `line` 将会是一个字符的字符串 `'\n'` （回车换行符）。 如果 `line` 是真的空字符串，就意味着文件已经没有行可读了。
3. 当我们到达文件尾时， 我们应关闭文件并抛出神奇的 `StopIteration` 异常。 记住，开门见山的说是因为我们需要为下一条规则找到一个匹配和应用功能。下一条规则从文件的下一行获取…… 但已经没有下一行了！ 所以，我们没有规则返回。 迭代器结束。 （♫ 派对结束 ♫）

由后往前直到 `__next__()` 方法的开始……

```
def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:

        if self.pattern_file.closed:

        .
        .
        .
```

1. `self.cache` 将是一个我们匹配并应用单独规则的功能列表。 （至少那个应该看起来熟悉！） `self.cache_index` 记录我们下一步返回的缓存条目。 如果我们还没有耗尽缓存 （举例 如果 `self.cache` 的长度大于 `self.cache_index` ）， 那么我们就会命中一条缓存！ 哇！ 我们可以从缓存中返回匹配和应用功能而不是从无到有创建。
2. 另一方面，如果我们没有从缓存中命中条目， 并且 文件对象也已关闭（这会发生， 在本方法下面一点， 正如你从预览的代码片段中所看到的），那么我们什么都不能做。 如果文件被关闭，意味着我们已经用完了它——我们已经从头至尾读取了模式文件的每一行，而且已经对每个模式创建并缓存了匹配和应用功能。文件已经读完；缓存已经用完；我也快完了。等等，什么？坚持一下，我们几乎完成了。

放到一起，发生了什么事？ 当：

- 当模块引入时，创建了 `LazyRules` 类的一个单一实例， 叫 `rules`， 它打开模式文件但

并没有读取。

- 当要求第一个匹配和应用功能时，检查缓存并发现缓存为空。 于是，从模式文件读取一行， 从模式中创建匹配和应用功能，并缓存之。
- 假如，因为参数的缘故，正好是第一行匹配了。如果那样，不会有更多的匹配和应用会创建，也不会有更多的行会从模式文件中读取。
- 更进一步， 因为参数的缘故，假设调用者再次调用 `plural()` 函数来让一个不同的单词变复数。 `plural()` 函数中的 `for` 循环会调用 `iter(rules)` ，这会重置缓存索引但不会重置打开的文件对象。
- 第一次遍历， `for` 循环会从 `rules` 中索要一个值，该值会调用其 `__next__()` 方法。然而这一次， 缓存已经被装入了一个匹配和应用功能对， 与模式文件中第一行模式一致。由于对前一个单词做复数变换时已经被创建和缓存，它们被从缓存中返回。 缓存索引递增，打开的文件无需访问。
- 假如，因为参数的缘故，这一轮第一个规则 不 匹配。 所以 `for` 循环再次运转并从 `rules` 请求一个值。 这会再次调用 `__next__()` 方法。 这一次， 缓存被用完了——它仅有一个条目，而我们被请求第二个——于是 `__next__()` 方法继续。 从打开的文件中读取下一行，从模式中创建匹配和应用功能，并缓存之。
- 该“读取创建并缓存”过程一直持续直到我们从模式文件中读取的规则与我们想变复数的单词不匹配。 如果我们确实在文件结束前找到了一个匹配规则，我们仅需使用它并停止，文件还一直打开。文件指针会留在我们停止读取，等待下一个 `readline()` 命令的地方。现在，缓存已经有更多条目了，并且再次从头开始来将一个新单词变复数，在读取模式文件下一行之前，缓存中的每一个条目都将被尝试。

我们已经到达复数变换的极乐世界。

1. 最小化初始代价。 在 `import` 时发生的唯一的事就是实例化一个单一的类并打开一个文件（但并不读取）。
2. 最大化性能 前述示例会在每次你想让一个单词变复数时，读遍文件并动态创建功能。本版本将在创建的同时缓存功能，在最坏情况下，仅需要读完一遍文件，无论你要让多少单词变复数。
3. 将代码和数据分离。 所有模式被存在一个分开的文件。代码是代码，数据是数据，二者永远不会交织。

> ☞这真的是极乐世界？嗯，是或不是。这里有一些 `LazyRules` 示例需要细想的地方：模式文件被打开（在 `__init__()` 中），并持续打开直到读取最后一个规则。当Python 退出或最后一个 `LazyRules` 类的实例销毁，Python 会最终关闭文件，但是那仍然可能会是一个很长的时间。如果该类是一个“长时间运行”的Python进程的一部分，Python可能从不退出，`LazyRules` 对象就可能一直不会释放。
>
> 这种情况有解决办法。不要在 `__init__()` 中打开文件并让其在一行一行读取规则时一直打开，你可以打开文件，读取所有规则，并立即关闭文件。或你可以打开文件，读取一条规则，用 `tell()` 方法保存文件位置，关闭文件，后面再次打开它，使用 `seek()` 方法 继续从你离开的地方读取。或者你不需担心这些就让文件打开，如同本示例所做。编程即是设计，而设计牵扯到所有的权衡和限制。让一个文件一直打开太长时间可能是问题；让你代码太复杂也可能是问题。哪一个是更大的问题，依赖于你的开发团队，你的应用，和你的运行环境。

深入阅读

- 迭代器类型
- PEP 234: 迭代器（ Iterators ）
- PEP 255: 简单生成器（ Simple Generators ）
- 系统程序员的生成器诀窍（ Generator Tricks for Systems Programmers ）

Chapter 8 高级迭代器

> " Great fleas have little fleas upon their backs to bite 'em, And little fleas have lesser fleas, and so ad infinitum. " — Augustus De Morgan

深入

H `AWAII + IDAHO + IOWA + OHIO == STATES`. 或者，换个说法，`510199 + 98153 + 9301 + 3593 == 621246`. 我在说是方言吗？不，这只是一个谜题。

让我来给你解释一下。

```
HAWAII + IDAHO + IOWA + OHIO == STATES
510199 + 98153 + 9301 + 3593 == 621246

H = 5
A = 1
W = 0
I = 9
D = 8
O = 3
S = 6
T = 2
E = 4
```

像这样的谜题被称为*cryptarithms* 或者 字母算术*(alphametics)*。字母可以拼出实际的单词，而如果你把每一个字母都用 `0-9` 中的某一个数字代替后, 也同样可以#8220;拼出" 一个算术等式。关键的地方是找出每个字母都映射到了哪个数字。每个字母所有出现的地方都必须映射到同一个数字，数字不能重复, 并且"单词"不能以0开始。

最著名的字母算术谜题是 `SEND + MORE = MONEY`。

在这一章中，我们将深入一个最初由Raymond Hettinger编写的难以置信的Python 程序。这个程序只用*14*行代码来解决字母算术谜题。

[下载 `alphametics.py`]

```
import re
import itertools

def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            if eval(equation):
                return equation

if __name__ == '__main__':
    import sys
    for puzzle in sys.argv[1:]:
        print(puzzle)
        solution = solve(puzzle)
        if solution:
            print(solution)
```

你可以从命令行运行这个程序。在Linux上, 运行情况看起来是这样的。(取决于你机器的速度，计算可能要花一些时间，而且不会有进度条。耐心等待就好了。)

```
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "HAWAII + IDAHO + IOWA +
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "I + LOVE + YOU == DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "SEND + MORE == MONEY"
SEND + MORE == MONEY
9567 + 1085 == 10652
```

找到一个模式所有出现的地方

字母算术谜题解决者做的第一件事是找到谜题中所有的字母(A–Z)。

```
>>> import re

['16', '2', '4', '8']

['SEND', 'MORE', 'MONEY']
```

1. `re` 模块是正则表达式的Python实现。它有一个漂亮的函数 `findall()`，接受一个正则表达式和一个字符串作为参数，然后找出字符串中出现该模式的所有地方。在这个例子里，模式匹配的是数字序列。 `findall()` 函数返回所有匹配该模式的子字符串的列表。
2. 这里正则表达式匹配的是字母序列。再一次，返回值是一个列表，其中的每一个元素是

匹配该正则表达式的字符串。

这是另外一个稍微复杂一点的例子。

```
>>> re.findall(' s.*? s', "The sixth sick sheikh's sixth sheep's sick.")
[' sixth s', " sheikh's s", " sheep's s"]
```

这是英语中最难的绕口令。

很惊奇？这个正则表达式寻找一个空格，一个 `s` , 然后是最短的任何字符构成的序列(`.*?`), 然后是一个空格, 然后是另一个 `s` 。 在输入字符串中，我看见了五个匹配:

1. `The <mark>sixth s</mark>ick sheikh's sixth sheep's sick.`
2. `The sixth <mark>sick s</mark>heikh's sixth sheep's sick.`
3. `The sixth sick <mark>sheikh's s</mark>ixth sheep's sick.`
4. `The sixth sick sheikh's <mark>sixth s</mark>heep's sick.`
5. `The sixth sick sheikh's sixth <mark>sheep's s</mark>ick.`

但是 `re.findall()` 函数值只返回了3个匹配。准确的说，它返回了第一，第三和第五个。为什么呢？因为它不会返回重叠的匹配。第一个匹配和第二个匹配是重叠的，所以第一个被返回了，第二个被跳过了。然后第三个和第四个重叠，所以第三个被返回了，第四个被跳过了。最后，第五个被返回了。三个匹配，不是五个。

这和字母算术解决者没有任何关系；我只是觉得这很有趣。

在序列中寻找不同的元素

Sets 使得在序列中查找不同的元素变得很简单。

```
>>> a_list = ['The', 'sixth', 'sick', "sheik's", 'sixth', "sheep's", 'sick']

{'sixth', 'The', "sheep's", 'sick', "sheik's"}
>>> a_string = 'EAST IS EAST'

{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']

'SENDMOREMONEY'

{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

1. 给出一个有若干字符串组成的列表， `set()` 函数返回列表中不同的字符串组成的集合。把它想象成一个 `for` 循环可以帮助理解。从列表出拿出第一个元素，放到集合。第二个，第三个，第四个。第五个，等等, 它已经在集合里面了，因为Python 集合不允许重复，所以它只被列出了一次。第六个。第七个又是一个重复的，所以它只被列出了一次。原来的列表甚至不需要事先排好序。
2. 同样的技术也适用于字符串，因为一个字符串就是一个字符序列。

3. 给出一个字符串列表，`''.join(` a_list `)` 将所有的字符串拼接成一个。
4. 所以，给出一个字符串列表，这行代码返回这些字符串中出现过的不重复的字符。

字母算术解决者通过这个技术来建立谜题中出现的不同字符的集合。

```
unique_characters = set(''.join(words))
```

这个列表在接下来迭代可能的解法的时候将被用来将数字分配给字符。

作出断言

和很多编程语言一样，Python 有一个 `assert` 语句。这是它的用法。

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Only for very large values of 2
```

1. `assert` 语句后面跟任何合法的Python 表达式。在这个例子里, 表达式 `1 + 1 == 2` 的求值结果为 `True` , 所以 `assert` 语句没有做任何事情。
2. 然而, 如果Python 表达式求值结果为 `False` , `assert` 语句会抛出一个 `AssertionError` .
3. 你可以提供一个人类可读的消息，`AssertionError` 异常被抛出的时候它可以被用于打印输出。

因此, 这行代码:

```
assert len(unique_characters) <= 10, 'Too many letters'
```

…等价于:

```
if len(unique_characters) > 10:
    raise AssertionError('Too many letters')
```

字母算术谜题使用这个 `assert` 语句来排除谜题包含多于10个的不同的字母的情况。因为每个不同的字母对应一个不同的数字，而数子只有10个,含有多于10个的不同的字母的谜题是不可能有解的。

生成器表达式

生成表达式类似生成器函数，只不过它不是函数。

```
>>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}

<generator object <genexpr> at 0x00BADC10>

69
>>> next(gen)
68

(69, 68, 77, 79, 78, 83, 82, 89)
```

1. 生成器表达式类似一个yield值的匿名函数。表达式本身看起来像列表解析, 但不是用方括号而是用圆括号包围起来。
2. 生成器表达式返回迭代器。
3. 调用 `next( gen )` 返回迭代器的下一个值。
4. 如果你愿意，你可以将生成器表达式传给 `tuple()` , `list()` , 或者 `set()` 来迭代所有的值并且返回元组，列表或者集合。在这种情况下，你不需要一对额外的括号 — 将生成器表达式 `ord(c) for c in unique_characters` 传给 `tuple()` 函数就可以了, Python 会推断出它是一个生成器表达式。

> ☞使用生成器表达式取代列表解析可以同时节省CPU 和 内存(RAM)。如果你构造一个列表的目的仅仅是传递给别的函数，(比如 传递给 `tuple()` 或者 `set()`), 用生成器表达式替代吧!

这里是到达同样目的的另一个方法, 使用生成器函数:

```
def ord_map(a_string):
    for c in a_string:
        yield ord(c)

gen = ord_map(unique_characters)
```

生成器表达式功能相同但更紧凑。

计算排列... 懒惰的方法!

首先, 排列到底是个什么东西? 排列是一个数学概念。(取决于你在处理哪种数学，排列有好几个定义。在这里我们说的是组合数学, 如果你完全不知道组合数学是什么也不用担心。同往常一样, 维基百科是你的朋友。)

想法是这样的，你有某物件(可以是数字，可以是字母，也可以是跳舞的熊)的一个列表，接着找出将它们拆开然后组合成小一点的列表的所有可能。所有的小列表的大小必须一致。最小是1，最大是元素的总数目。哦，也不能有重复。数学家说"让我们找出3个元素取2个的排列," 意思是你有一个3个元素的序列，然后你找出所有可能的有序对。

```
(1, 2)
>>> next(perms)
(1, 3)
>>> next(perms)

>>> next(perms)
(2, 3)
>>> next(perms)
(3, 1)
>>> next(perms)
(3, 2)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

1. `itertools` 模块里有各种各样的有趣的东西，包括 `permutations()` 函数，它把查找排列的所有辛苦的工作的做了。
2. `permutations()` 函数接受一个序列(这里是3个数字组成的列表) 和一个表示你要的排列的元素的数目的数字。函数返回迭代器，你可以在 `for` 循环或其他老地方使用它。这里我遍历迭代器来显示所有的值。
3. `[1, 2, 3]` 取2个的第一个排列是 `(1, 2)` 。
4. 记住排列是有序的: `(2, 1)` 和 `(1, 2)` 是不同的。
5. 这就是了。这些就是 `[1, 2, 3]` 取两个的所有排列。像 `(1, 1)` 或者 `(2, 2)` 这样的元素对没有出现，因为它们包含重复导致它们不是合法的排列。当没有更多排列的时候，迭代器抛出一个 `StopIteration` 异常。

`itertools` 模块有各种各样的有趣的东西。

`permutations()` 函数并不一定要接受列表。它接受任何序列 — 甚至是字符串。

```
>>> import itertools

>>> next(perms)

>>> next(perms)
('A', 'C', 'B')
>>> next(perms)
('B', 'A', 'C')
>>> next(perms)
('B', 'C', 'A')
>>> next(perms)
('C', 'A', 'B')
>>> next(perms)
('C', 'B', 'A')
>>> next(perms)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

[('A', 'B', 'C'), ('A', 'C', 'B'),
 ('B', 'A', 'C'), ('B', 'C', 'A'),
 ('C', 'A', 'B'), ('C', 'B', 'A')]
```

1. 字符串就是一个字符序列。对于查找排列来说，字符串 `'ABC'` 和列表 `['A', 'B', 'C']` 是等价的。

2. `['A', 'B', 'C']` 取3个的第一个排列是 `('A', 'B', 'C')` 。还有5个其他的排列 — 同样的3个字符，不同的顺序。
3. 由于 `permutations()` 函数总是返回迭代器，一个简单的调试排列的方法是将这个迭代器传给内建的 `list()` 函数来立刻看见所有的排列。

`itertools` 模块中的其它有趣的东西

```
>>> import itertools

[('A', '1'), ('A', '2'), ('A', '3'),
 ('B', '1'), ('B', '2'), ('B', '3'),
 ('C', '1'), ('C', '2'), ('C', '3')]

[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

1. `itertools.product()` 函数返回包含两个序列的笛卡尔乘积的迭代器。
2. `itertools.combinations()` 函数返回包含给定序列的给定长度的所有组合的迭代器。这和 `itertools.permutations()` 函数很类似，除了不包含因为只有顺序不同而重复的情况。所以 `itertools.permutations('ABC', 2)` 同时返回 `('A', 'B')` and `('B', 'A')` (同其它的排列一起)，`itertools.combinations('ABC', 2)` 不会返回 `('B', 'A')` ，因为它和 `('A', 'B')` 是重复的，只是顺序不同而已。

[下载 `favorite-people.txt`]

```
>>> names
['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n',
'Mike\n', 'Chris\n', 'Sarah\n', 'Alex\n', 'Lizzie\n']

>>> names
['Dora', 'Ethan', 'Wesley', 'John', 'Anne',
'Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']

>>> names
['Alex', 'Anne', 'Chris', 'Dora', 'Ethan',
'John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']

>>> names
['Alex', 'Anne', 'Dora', 'John', 'Mike',
'Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']
```

1. 这个表达式将文本内容以一行一行组成的列表的形式返回。
2. 不幸的是，(对于这个例子来说)，`list(open(` filename `))` 表达式返回的每一行的末尾都包含回车。这个列表解析使用 `rstrip()` 字符串方法移除每一行尾部的空白。(字符串也有一个 `lstrip()` 方法移除头部的空白，以及 `strip()` 方法头尾都移除。)
3. `sorted()` 函数接受一个列表并将它排序后返回。默认情况下，它按字母序排序。
4. 然而， `sorted()` 函数也接受一个函数作为 `key` 参数, 并且使用key来排序。在这个例子里，排序函数是 `len()` ,所以它按 `len(` each item `)` 来排序。短的名字排在前面，然后是稍长，接着是更长的。

这和 `itertools` 模块有什么关系? 很高兴你问了这个问题。

```
…continuing from the previous interactive shell…
>>> import itertools

>>> groups
<itertools.groupby object at 0x00BB20C0>
>>> list(groups)
[(4, <itertools._grouper object at 0x00BA8BF0>),
 (5, <itertools._grouper object at 0x00BB4050>),
 (6, <itertools._grouper object at 0x00BB4030>)]

...     print('Names with {0:d} letters:'.format(name_length))
...     for name in name_iter:
...         print(name)
...
Names with 4 letters:
Alex
Anne
Dora
John
Mike
Names with 5 letters:
Chris
Ethan
Sarah
Names with 6 letters:
Lizzie
Wesley
```

1. `itertools.groupby()` 函数接受一个序列和一个key 函数, 并且返回一个生成二元组的迭代器。每一个二元组包含 `key_function(` each item `)` 的结果和另一个包含着所有共享这个key结果的元素的迭代器。
2. 调用 `list()` 函数会"耗尽"这个迭代器, 也就是说 你生成了迭代器中所有元素才创造了这个列表。迭代器没有"重置"按钮。你一旦耗尽了它，你没法重新开始。如果你想要再循环一次(例如, 在接下去的 `for` 循环里面), 你得调用 `itertools.groupby()` 来创建一个新的迭代器。
3. 在这个例子里，给出一个已经按长度排序的名字列表，`itertools.groupby(names, len)` 将会将所有的4个字母的名字放在一个迭代器里面，所有的5个字母的名字放在另一个迭代器里，以此类推。 `groupby()` 函数是完全通用的; 它可以将字符串按首字母，将数字按因子数目, 或者任何你能想到的key函数进行分组。

> ☞ `itertools.groupby()` 只有当输入序列已经按分组函数排过序才能正常工作。在上面的例子里面，你用 `len()` 函数分组了名字列表。这能工作是因为输入列表已经按长度排过序了。

Are you watching closely?

```
>>> list(range(0, 3))
[0, 1, 2]
>>> list(range(10, 13))
[10, 11, 12]

[0, 1, 2, 10, 11, 12]

[(0, 10), (1, 11), (2, 12)]

[(0, 10), (1, 11), (2, 12)]

[(0, 10), (1, 11), (2, 12), (None, 13)]
```

1. `itertools.chain()` 函数接受两个迭代器，返回一个迭代器，它包含第一个迭代器的所有内容，以及跟在后面的来自第二个迭代器的所有内容。(实际上，它接受任何数目的迭代器，并把它们按传入顺序串在一起。)
2. `zip()` 函数的作用不是很常见，结果它却非常有用: 它接受任何数目的序列然后返回一个迭代器，其第一个元素是每个序列的第一个元素组成的元组，然后是每个序列的第二个元素（组成的元组），以此类推。
3. `zip()` 在到达最短的序列结尾的时候停止。 `range(10, 14)` 有四个元素(10, 11, 12, 和13), 但是 `range(0, 3)` 只有3个, 所以 `zip()` 函数返回包含3个元素的迭代器。
4. 相反， `itertools.zip_longest()` 函数在到达最长的序列的结尾的时候才停止, 对短序列结尾之后的元素填入 `None` 值.

好吧，这些都很有趣，但是和字母算术谜题解决者有什么联系呢? 请看下面:

```
>>> characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
>>> guess = ('1', '2', '0', '3', '4', '5', '6', '7')

(('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'),
 ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '7'))

{'E': '0', 'D': '3', 'M': '2', 'O': '4',
 'N': '5', 'S': '1', 'R': '6', 'Y': '7'}
```

1. 给出一个字母列表和一个数字列表(两者的元素的形式都是1个字符的字符串), `zip` 函数按顺序创建一组组字母，数字对。
2. 为什么这很酷? 因为这个数据结构正好可以用来传递给 `dict()` 函数来创建以字母为键，对应数字为值的字典。(这不是实现这个目的唯一方法。你当然可以使用字典解析来直接创建字典。) 尽管字典的打印形式以另一个顺序列出了这些键值对(字典本身没有#8220;顺序"), 但是你可以看见每一个字母都按 `characters` 和 `guess` 序列的原始顺序对应到了相应的数字。

算术谜题解决者使用这个技术对每一个可能的解法创建一个将谜题中的字母映射到解法中的数字的字典。

```
characters = tuple(ord(c) for c in sorted_characters)
digits = tuple(ord(c) for c in '0123456789')
...
for guess in itertools.permutations(digits, len(characters)):
    ...
 <mark>equation = puzzle.translate(dict(zip(characters, guess)))</mark>
```

但是 `translate()` 方法是什么呢? 啊哈, 我们现在到了真正有趣的部分了。

一种新的操作字符串的方法

Python 字符串有很多方法。我们在字符串章节中学习了其中一些: `lower()` , `count()` , 和 `format()` 。现在我要给你介绍一个强大但鲜为人知的操作字符串的技术: `translate()` 方法。

```
{65: 79}

'MORK'
```

1. 字符串翻译从一个转换表开始, 转换表就是一个将一个字符映射到另一个字符的字典。实际上, "字符" 是不正确的 — 转换表实际上是将一个 字节 (*byte*)映射到另一个。
2. 记住, Python 3 中的字节是整形数。 `ord()` 函数返回字符的ASCII码。在这个例子中, 字符是A–Z, 所以返回的是从65 到 90的字节。
3. 一个字符串的 `translate()` 方法接收一个转换表, 并用它来转换该字符串。换句话说, 它将出现在转换表的键中的字节替换为该键对应的值。在这个例子里, 将 `MARK` "翻译为" `MORK` .

现在你开始进入真正有趣的部分了。

这和解决字母算术谜题有什么关系呢？实际上, 关系大着呢。

```
>>> characters
(83, 77, 69, 68, 79, 78, 82, 89)

>>> guess
(57, 49, 53, 55, 48, 54, 56, 50)

>>> translation_table
{68: 55, 69: 53, 77: 49, 78: 54, 79: 48, 82: 56, 83: 57, 89: 50}

'9567 + 1085 == 10652'
```

1. 使用生成器表达式, 我们快速的计算出字符串中每个字符的字节值。 `characters` 是 `alphametics.solve()` 函数中的 `sorted_characters` 的示例值 .
2. 使用另一个生成器表达式, 我们快速的计算出字符串中每个数字的字节值。计算结果 `guess` , 正好是 `alphametics.solve()` 函数中的 `itertools.permutations()` 函数返回值的

格式。
3. 通过将 `characters` 和 `guess` zipping 出来的元素对序列构造出的字典来作为转换表。这正是 `alphametics.solve()` 在 `for` 循环里面干的事情。
4. 最后我们将转换表传递给原始字符串的 `translate()` 方法。这会将字符串中的每个字母转化成相应的数字(基于 `characters` 中字母和 `guess` 中的数字)。结果是一个字符串形式的合法的Python表达式。

这相当令人难忘。但你能对正巧是一个合法Python 表达式的字符串干什么呢？

将任何字符串作为Python表达式求值

这是谜题的最后一部分(或者说, 谜题解决者的最后一部分)。经过华丽的字符串操作，我们得到了类似 `'9567 + 1085 == 10652'` 这样的一个字符串。但那是一个字符串，字符串有什么好的？输入 `eval()` , Python 通用求值工具。

```
>>> eval('1 + 1 == 2')
True
>>> eval('1 + 1 == 3')
False
>>> eval('9567 + 1085 == 10652')
True
```

但是等一下，不止这些! `eval()` 并不限于布尔表达式。它能处理任何 Python 表达式并且返回任何数据类型。

```
>>> eval('"A" + "B"')
'AB'
>>> eval('"MARK".translate({65: 79})')
'MORK'
>>> eval('"AAAAA".count("A")')
5
>>> eval('["*"] * 5')
['*', '*', '*', '*', '*']
```

等一下，还没完呢!

```
>>> x = 5

25

25
>>> import math

2.2360679774997898
```

1. `eval()` 接受的表达式可以引用在 `eval()` 之外定义的全局变量。如果(`eval()`)在函数内被调用, 它也可以引用局部变量。
2. 以及函数。
3. 以及模块。

喂，等一下...

```
>>> import subprocess

'Desktop         Library         Pictures \
 Documents       Movies          Public   \
 Music           Sites'
```

1. `subprocess` 模块允许你执行任何shell命令并以字符串形式获得输出。
2. 执行任意的shell命令可能会导致永久的（不好的）后果。

更坏的是，由于存在全局函数 `__import__()` ，它接收字符串形式的模块名，导入模块，并返回模块的引用。和 `eval()` 的能力结合起来，你可以构造一个单独的表达式来删除你所有的文件:

1. 现在想象一下 `'rm -rf ~'` 的输出。实际上它不会有任何输出，但是你也不会有任何文件还留着。

eval() 是邪恶的

好吧, 邪恶部分是对来自非信任源的表达式进行求值。你应该只在信任的输入上使用 `eval()` 。当然，关键的部分是确定什么是"可信任的"。但有一点我敢肯定: 你不应该将这个字母算术表达式放到网上最为一个小的web服务。不要错误的认为，"Gosh, 这个函数在求值以前做了那么多的字符串操作。我想不出 谁能利用这个漏洞。" 会有人找出穿过这些字符串操作把危险的可执行代码放进来的方法的。(更奇怪的事情都发生过。), 然后你就得和你的服务器说再见了。

但是肯定有某种办法可以安全的求值表达式吧？将 `eval()` 放到一个不能访问和伤害外部世界的沙盒里面。嗯，对也不对。

```
>>> x = 5

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined

>>> import math

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'math' is not defined
```

1. 传给 `eval()` 函数的第二和第三个函数担当了求值表达式是的全局和局部名字空间的角色。在这个例子里，它们都是空的，意味着当字符串 `"x * 5"` 被求值的时候, 在全局和本地的名字空间都没有变量 `x` , 所以 `eval()` 抛出了一个异常。
2. 你可以通过一个个列出的方式选择性在全局名字空间里面包含一些值。这些 — 并且这有这些 — 变量在求值的时候可用。
3. 即使你刚刚导入了 `math` 模块, 你没有在传给 `eval()` 函数的名字空间里包含它，所以求值

失败了。

哎呀，这很简单。 让我来做一个字母算术谜题的Web服务吧！

```
25

2.2360679774997898
```

1. 即使你传入空的字典作为全局和局部名字空间，所有的Python 内建函数在求值时还是可用的。所以 `pow(5, 2)` 可以工作, 因为 `5` 和 `2` 是字面量，而 `pow()` 是内建函数。
2. 很不幸 (如果你不明白为什么不幸，继续读。), `__import__()` 也是一个内建函数，所以它也能工作。

是的，这意味着即使你在调用 `eval()` 的时候显式的将全局和局部名字空间设置为空字典，你仍然可以做坏事。

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')", {}, {})
```

哎呀. 幸亏我没有做那个字母算术web服务。存在任何安全的使用 `eval()` 的方法吗? 嗯, 有也没有。

```
>>> eval("__import__('math').sqrt(5)",

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
>>> eval("__import__('subprocess').getoutput('rm -rf /')",

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

1. 为了安全的求值不受信任的表达式, 你需要定义一个将 `"__builtins__"` 映射为 `None` (Python 的空值)的全局名字空间字典. 在内部, "内建" 函数包含在一个叫做 `"__builtins__"` 的伪模块内。这个伪模块(即 内建函数的集合) 在没有被你显式的覆盖的情况下对被求值的表达式是总是可用的。
2. 请确保你覆盖的是 `__builtins__` 。 不是 `__builtin__` , `__built-ins__` , 或者其它某个变量，否则程序还是可以运行但是会有巨大的风险。

那么 `eval()` 现在安全了? 嗯，是也不是。

```
>>> eval("2 ** 2147483647",
```

1. 即使不能访问到 `__builtins__` , 你还是可以开启一个拒绝服务攻击。例如, 试图求 `2` 的 `2147483647` 次方会导致你的服务器的 CPU 利用率到达100% 一段时间。(如果你在交互

式shell中试验这个, 请多按几次 `Ctrl-C` 来跳出来。) 技术上讲，这个表达式 最终将会返回一个值, 但是在这段时间里你的服务器将啥也干不了。

最后, Python 表达式的求值是可能达到某种意义的“安全”的, 但结果是在现实生活中没什么用。如果你只是玩玩没有问题，如果你只给它传递安全的输入也没有问题。但是其它的情况完全是自找麻烦。

把所有东西放在一起

总的来说: 这个程序通过暴力解决字母算术谜题， 也就是通过穷举所有可能的解法。为了达到目的，它

1. 通过 `re.findall()` 函数找到谜题中的所有字母
2. 使用集合和 `set()` 函数找到谜题出现的所有不同的字母
3. 通过 `assert` 语句检查是否有超过10个的不同的字母 (意味着谜题无解)
4. 通过一个生成器对象将字符转换成对应的ASCII码值
5. 使用 `itertools.permutations()` 函数计算所有可能的解法
6. 使用 `translate()` 字符串方法将所有可能的解转换成Python表达式
7. 使用 `eval()` 函数通过求值Python 表达式来检验解法
8. 返回第一个求值结果为 `True` 的解法

…仅仅14行代码.

进一步阅读

- `itertools` 模块
- `itertools` — 用于高效循环的迭代器函数
- 观看 Raymond Hettinger 在 PyCon 2009 上的 “Easy AI with Python” 演讲
- Recipe 576615: Alphametics solver, Raymond Hettinger 的原始的适用于Python 2的算木谜题解决程序
- More of Raymond Hettinger’s recipes in the ActiveState Code repository
- 算木谜题在维基百科上的页面
- 字母索引, 包含 很多谜题 以及 一个创建你自己的谜题的工具

非常感谢Raymond Hettinger同意重现授权他的代码，因此我才能将它移植到Python 3 并作为本章的基础。

Chapter 9 单元测试

> " Certitude is not the test of certainty. We have been cocksure of many things that were not so. " — Oliver Wendell Holmes, Jr.

（不要）深入

在此章节中，你将要编写及调试一系列用于阿拉伯数字与罗马数字相互转换的方法。你阅读了在“案例学习：罗马数字”中关于构建及校验罗马数字的机制。那么，现在考虑扩展该机制为一个双向的方法。

罗马数字的规则引出很多有意思的结果：

1. 只有一种正确的途径用阿拉伯数字表示罗马数字。
2. 反过来一样，一个字符串类型的有效的罗马数字也仅可以表示一个阿拉伯数字（即，这种转换方式也是只有一种）。
3. 只有有限范围的阿拉伯数字可以以罗马数字表示，那就是1-3999。而罗马数字表示大数字却有几种方式。例如，为了表示一个数字连续出现时正确的值则需要乘以 `1000` 。为了达到本节的目的，限定罗马数字在 1 到 3999 之间。
4. 无法用罗马数字来表示 0 。
5. 无法用罗马数字来表示负数 。
6. 无法用罗马数字来表示分数或非整数 。

现在，开始设计 `roman.py` 模块。它有两个主要的方法： `to_roman()` 及 `from_roman()` 。 `to_roman()` 方法接收一个从 `1` 到 `3999` 之间的整型数字，然后返回一个字符串类型的罗马数字。

在这里停下来。现在让我们进行一些意想不到的操作：编写一个测试用例来检测 `to_roman` 函数是否实现了你想要的功能。你想得没错：你正在编写测试尚未编写代码的代码。

这就是所谓的测试驱动开发 或 TDD。那两个转换方法（ `to_roman()` 及之后的 `from_roman()` ）可以独立于任何使用它们的大程序而作为一个单元来被编写及测试。Python 自带一个单元测试框架，被恰当地命名为 `unittest` 模块。

单元测试是整个以测试为中心的开发策略中的一个重要部分。编写单元测试应该安排在项目的早期，同时要让它随同代码及需求变更一起更新。很多人都坚持测试代码应该先于被测试代码的，而这种风格也是我在本节中所主张的。但是，不管你何时编写，单元测试都是有好处的。

- 在编写代码之前，通过编写单元测试来强迫你使用有用的方式细化你的需求。
- 在编写代码时，单元测试可以使你避免过度编码。当所有测试用例通过时，实现的方法

就完成了。

- 重构代码时，单元测试用例有助于证明新版本的代码跟老版本功能是一致的。
- 在维护代码期间，如果有人对你大喊：你最新的代码修改破坏了原有代码的状态，那么此时单元测试可以帮助你反驳（“先生，所有单元测试用例通过了我才提交代码的...”）。
- 在团队编码中，缜密的测试套件可以降低你的代码影响别人代码的机会，这是因为你需要优先执行别人的单元测试用例。（我曾经在代码冲刺见过这种实践。一个团队把任务分解，每个人领取其中一小部分任务，同时为其编写单元测试;然后，团队相互分享他们的单元测试用例。这样，所有人都可以在编码过程中提前发现谁的代码与其他人的不可以良好工作。）

一个简单的问题

每个测试都是一个孤岛。

一个测试用例仅回答一个关于它正在测试的代码问题。一个测试用例应该可以：

- ……完全自动运行，而不需要人工干预。单元测试几乎是全自动的。
- ……自主判断被测试的方法是通过还是失败，而不需要人工解释结果。
- ……独立运行，而不依赖其它测试用例（即使测试的是同样的方法）。即，每一个测试用例都是一个孤岛。

让我们据此为第一个需求建立一个测试用例：

1. `to_roman()` 方法应该返回代表 `1` - `3999` 的罗马数字。

这些代码功效如何并不那么显而易见。它定义了一个没有 `__init__` 方法的类。而该类当然有其它方法，但是这些方法都不会被调用。在整个脚本中，有一个**main** 块，但它并不引用该类及它的方法。但我承诺，它做别的事情了。

```
import roman1
import unittest

    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
                     (1110, 'MCX'),
                     (1226, 'MCCXXVI'),
                     (1301, 'MCCCI'),
                     (1485, 'MCDLXXXV'),
                     (1509, 'MDIX'),
                     (1607, 'MDCVII'),
                     (1754, 'MDCCLIV'),
                     (1832, 'MDCCCXXXII'),
                     (1993, 'MCMXCIII'),
                     (2074, 'MMLXXIV'),
                     (2152, 'MMCLII'),
                     (2212, 'MMCCXII'),
                     (2343, 'MMCCCXLIII'),
                     (2499, 'MMCDXCIX'),
                     (2574, 'MMDLXXIV'),
                     (2646, 'MMDCXLVI'),
                     (2723, 'MMDCCXXIII'),
                     (2892, 'MMDCCCXCII'),
                     (2975, 'MMCMLXXV'),
                     (3051, 'MMMLI'),
                     (3185, 'MMMCLXXXV'),
                     (3250, 'MMMCCL'),
                     (3313, 'MMMCCCXIII'),
                     (3408, 'MMMCDVIII'),
                     (3501, 'MMMDI'),
                     (3610, 'MMMDCX'),
                     (3743, 'MMMDCCXLIII'),
                     (3844, 'MMMDCCCXLIV'),
                     (3888, 'MMMDCCCLXXXVIII'),
                     (3940, 'MMMCMXL'),

        '''to_roman should give known result with known input'''
        for integer, numeral in self.known_values:

if __name__ == '__main__':
    unittest.main()
```

1. 为了编写测试用例，首先使该测试用例类成为 `unittest` 模块的 `TestCase` 类的子类。TestCase 提供了很多你可以用于测试特定条件的测试用例的有用的方法。

2. 这是一张我手工核实过的整型数字-罗马数字对的列表。它包括最小的十个数字、最大数字、每一个有唯一一个字符串格式的罗马数字的数字以及一个有其它有效数字产生的随机数。你没有必要测试每一个可能的输入，而需要测试所有明显的边界用例。
3. 每一个独立的测试都有它自己的不含参数及没有返回值的方法。如果方法不抛出异常而正常退出则认为测试通过;否则，测试失败。
4. 这里调用了真实的 `to_roman()` 方法. （当然，该方法还没编写;但一旦该方法被实现，这就是调用它的行号）。注意，现在你已经为 `to_roman()` 方法定义了 接口：它必须包含一个整型（被转换的数字）及返回一个字符串（罗马数字的表示形式）。如果 接口 实现与这些定义不一致，那么测试就会被视为失败。同样，当你调用 `to_roman()` 时，不要捕获任何异常。这些都是unittest 故意设计的。当你以有效的输入调用 `to_roman()` 时它不会抛出异常。如果 `to_roman()` 抛出了异常，则测试被视为失败。
5. 假设 `to_roman()` 方法已经被正确定义，正确调用，成功实现以及返回了一个值，那么最后一步就是去检查它的返回值是否 *right* 。这是测试中一个普遍的问题。 `TestCase` 类提供了一个方法 `assertEqual` 来检查两个值是否相等。如果 `to_roman()` (`result`) 的返回值跟已知的期望值g (`numeral`)不一致，则抛出异常，并且测试失败。如果两值相等， `assertEqual` 不会做任何事情。如果 `to_roman()` 的所有返回值均与已知的期望值一致，则 `assertEqual` 不会抛出任何异常，于是， `test_to_roman_known_values` 最终会会正常退出，这就意味着 `to_roman()` 通过此次测试。

编写一个失败的测试，然后进行编码直到该测试通过。

一旦你有了测试用例，你就可以开始编写 `to_roman()` 方法。首先，你应该用一个空方法作为存根，同时确认该测试失败。因为如果在编写任何代码之前测试已经通过，那么你的测试对你的代码是完全不会有效果的！单元测试就像跳舞：测试先行，编码跟随。编写一个失败的测试，然后进行编码直到该测试通过。

```
# roman1.py

def to_roman(n):
    '''convert integer to Roman numeral'''
```

1. 在此阶段，你想定义to_roman()方法的 API ，但是你还不想编写（首先，你的测试需要失败）。为了存根，需要使用Python 保留关键字 `pass` ，它恰恰什么都没做。

在命令行上运行 `romantest1.py` 来执行该测试。如果使用-v命令行参数的话，会有更详细的输出来帮助你精确地查看每一条用例的执行过程。幸运的话，你的输出应该如下：

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v

 ======================================================================
FAIL: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest1.py", line 73, in test_to_roman_known_values
    self.assertEqual(numeral, result)

----------------------------------------------------------------------
```

1. 运行脚本就会执行 `unittest.main()` ,该方法执行了每一条测试用例。而每一条测试用例都是 `romantest.py` 中的类方法。这些测试类没有必要的组织要求;它们每一个都包括一个独立的测试方法，或者你也可以编写一个含有多个测试方法的类。唯一的要求就是每一个测试类都必须继承 `unittest.TestCase` 。
2. 对于每一个测试用例， `unittest` 模块会打印出测试方法的 `docstring` ，并且说明该测试失败还是成功。正如预期那样，该测试用例失败了。
3. 对于每一个失败的测试用例， `unittest` 模块会打印出详细的跟踪信息。在该用例中， `assertEqual()` 的调用抛出了一个 `AssertionError` 的异常，这是因为 `to_roman(1)` 本应该返回 `'I'` 的，但是它没有。（因为没有显示的返回值，故方法返回了 Python 的空值 `None` ）
4. 在说明每个用例的详细执行结果之后， `unittest` 打印出一个简述来说明“多少用例被执行了”和“测试执行了多长时间”。
5. 从整体上说，该测试执行失败，因为至少有一条用例没有成功。如果测试用例没有通过的话， `unittest` 可以区别用例执行失败跟程序错误的。像 `assertXYZ` 、 `assertRaises` 这样的 `assertEqual` 方法的失败是因为被声明的条件不是为真，或者预期的异常没有抛出。错误，则是另一种异常，它是因为被测试的代码或者单元测试用例本身的代码问题而引起的。

至此，你可以实现 `to_roman()` 方法了。

```
roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),

def to_roman(n):
    '''convert integer to Roman numeral'''
    result = ''
    for numeral, integer in roman_numeral_map:

            result += numeral
            n -= integer
    return result
```

1. `roman_numeral_map` 是一个由元组组成的元组，它定义了三样东西：代表最基本的罗马数字的字符、罗马数字的顺序（逆序，从 `M` 到 `I` ）、每一个罗马数字的阿拉伯数值。每一个内部的元组都是一个 `(` 数 `,` 值 `)` 对。它不但定义了单字符罗马数字，也定义了双字符罗马数字，如 `CM` （“比一千小一百”）。该元组使得 `to_roman()` 方法实现起来更简单。
2. 这里得益于 `roman_numeral_map` 的数据结构，因为你不需要任何特别得逻辑去处理减法。为了转化成罗马数字，通过查找等于或者小于输入值的最大值来简化对

`roman_numeral_map` 的迭代。一旦找到，就把罗马数字的字符串追加至输出值（result）末段，同时输入值要减去相应的数值，如此重复。

如果你仍然不清楚 `to_roman()` 如何工作，可以在 `while` 循环末段添加 `print()` 调用：

```
while n >= integer:
    result += numeral
    n -= integer
    print('subtracting {0} from input, adding {1} to output'.format(integer, numeral))
```

因为用于调试的 `print()` 声明，输出会如下：

```
>>> import roman1
>>> roman1.to_roman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

这样， `to_roman()` 至少在手工检查下是工作正常的。但它会通过你编写的测试用例么？

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)

----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

1. 万岁！ `to_roman()` 函数通过了“known values” 测试用例。该测试用例并不复杂，但是它的确使该方法按着输入值的变化而执行，其中的输入值包括：每一个单字符罗马数字、最大值数字（ `3999` ）、最长字符串数字（ `3888` ）。通过这些，你就可以有理由对“该方法接收任何正常的输入值都工作正常”充满信心了。

“正常”输入？”嗯。那“非法”输入呢？

“停止然后着火”

Python 方式的停止并点火实际是引发一个例外。

仅仅在“正常”值时证明方法通过的测试是不够的;你同样需要测试当输入“非法”值时方法失败。但并不是说要枚举所有的失败类型，而是说必要在你预期的范围内失败。

```
>>> import roman1
>>> roman1.to_roman(4000)
'MMMM'
>>> roman1.to_roman(5000)
'MMMMM'

'MMMMMMMMM'
```

1. 这明显不是你所期望的——那也不是一个合法的罗马数字！事实上，这些输入值都超过了允许的范围，但该函数却返回了假值。悄悄返回的错误值是 很糟糕 的，因为如果一个程序要挂掉的话，迅速且引人注目地挂掉会好很多。正如谚语"停止然后着火"。Python 方式的停止并点火实际是引发一个例外。

那问题是：我该如何表达这些内容为可测试需求呢？下面就是一个开始：

> 当输入值大于 `3999` 时， `to_roman()` 函数应该抛出一个 `OutOfRangeError` 异常。

具体测试代码如下：

```
        '''to_roman should fail with large input'''
```

1. 如前一个测试用例，创建一个继承于 `unittest.TestCase` 的类。你可以在每个类中实现多个测试（正如你在本节中将会看到的一样），但是我却选择了创建一个新类，因为该测试与上一个有点不同。这样，我们可以把正常输入的测试跟非法输入的测试分别放入不同的两个类中。
2. 如前一个测试用例，测试本身是类一个方法，并且该方法以 `test` 开头命名。
3. `unittest.TestCase` 类提供e `assertRaises` 方法，该方法需要以下参数：你期望的异常、你要测试的方法及传入给方法的参数。（如果被测试的方法需要多个参数的话，则把所有参数依次传入 `assertRaises` ， assertRaises 会正确地把参数传递给被测方法的。）

请关注代码的最后一行。这里并不需要直接调用 `to_roman()` ，同时也不需要手动检查它抛出的异常类型（通过 一个 `try...except` 块来包装），而这些 `assertRaises` 方法都给我们完成了。你要做的所有事情就是告诉assertRaises你期望的异常类型（ `roman2.OutOfRangeError` ）、被测方法（ `to_roman()` ）以及方法的参数（ `4000` ）。 `assertRaises` 方法负责调用 `to_roman()` 和检查方法抛出 `roman2.OutOfRangeError` 的异常。

另外，注意你是把 `to_roman()` 方法作为参数传递;你没有调用被测方法，也不是把被测方法作为一个字符串名字传递进去。我是否在之前提到过 Python 中万物皆对象有多么轻便？

那么，当你执行该含有新测试的测试套件时，结果如下：

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)

======================================================================
ERROR: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

1. 测试本应该是失败的（因为并没有任何代码使它通过），但是它没有真正的“失败”，而是出现了“错误”。这里有些微妙但是重要的区别。单元测试事实上有 三种 返回值：通过、失败以及错误。“通过”，但当然就是说测试成功了——被测代码符合你的预期。“失败”就是就如之前的测试用例一样（直到你编写代码令它通过）——执行了被测试的代码但返回值并不是所期望的。“错误”就是被测试的代码甚至没有正确执行。
2. 为什么代码没有正确执行呢？回溯说明了一切。你正在测试的模块没有叫 `OutOfRangeError` 的异常。回忆一下，该异常是你传递给 `assertRaises()` 方法的，因为你期望当传递给被测试方法一个超大值时可以抛出该异常。但是，该异常并不存在，因此 `assertRaises()` 的调用会失败。事实上测试代码并没有机会测试 `to_roman()` 方法，因为它还没有到达那一步。

为了解决该问题，你需要在 `roman2.py` 中定义 `OutOfRangeError` 。

1. 异常也是类。“越界”错误是值错误的一类——参数值超出了可接受的范围。所以，该异常继承了内建的 `ValueError` 异常类。这并不是严格的要求（它同样也可以继承于基类 `Exception` ），只要它正确就行了。
2. 事实上，异常类可以不做任何事情，但是至少添加一行代码使其成为一个类。 `pass` 的真正意思是什么都不做，但是它是一行Python代码，所以可以使其成为类。

再次执行该测试套件。

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)

======================================================================
FAIL: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)

----------------------------------------------------------------------
Ran 2 tests in 0.016s

FAILED (failures=1)
```

1. 新的测试仍然没有通过，但是它并没有返回错误而是失败。相反，测试失败了。这就是进步！它意味着这回 `assertRaises()` 方法的调用是成功的，同时，单元测试框架事实上也测试了 `to_roman()` 函数。
2. 当然 `to_roman()` 方法没有引发你所定义的 `OutOfRangeError` 异常，因为你并没有让它这么做。这真是个好消息！因为它意味着这是个合格的测试案例——在编写代码使之通过之前它将会以失败为结果。

现在可以编写代码使其通过了。

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if n > 3999:

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

1. 非常直观：如果给定的输入 (`n`) 大于 `3999` ，引发一个 `OutOfRangeError` 例外。本单元测试并不检测那些与例外相伴的人类可读的字符串，但你可以编写另一个测试来检查它（但请注意用户的语言或环境导致的不同国际化问题）。

这样能让测试通过吗？让我们来寻找答案。

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

1. 万岁！两个测试都通过了。因为你是在测试与编码之间来回反复开发的，所以你可以肯定使得其中一个测试从“失败”转变为“通过”的原因就是你刚才新添的两行代码。虽然这种信心来得并不简单，但是这种代价会在你代码的生命周期中得到回报。

More Halting, More Fire

与测试超大值一样，也必须测试超小值。正如我们在功能需求中提到的那样，罗马数字无法表达 0 或负数。

```
>>> import roman2
>>> roman2.to_roman(0)
''
>>> roman2.to_roman(-1)
''
```

显然，这不是好的结果。让我们为这些条件逐条添加测试。

```
class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''

    def test_zero(self):
        '''to_roman should fail with 0 input'''

    def test_negative(self):
        '''to_roman should fail with negative input'''
```

1. `test_too_large()` 方法跟之前的步骤一样。我把它包含进来是为了说明新代码的位置。
2. 这里是新的测试方法： `test_zero()` 。如 `test_too_large()` 一样，它调用了在n `unittest.TestCase` 中定义的 `assertRaises()` 方法，并且以参数值 0 传入给 `to_roman()` ，最后检查它抛出相应的异常： `OutOfRangeError` 。
3. `test_negative()` 也几乎类似，除了它给 `to_roman()` 函数传入 `-1` 。如果新的测试中没有 任何一个抛出了异常 `OutOfRangeError` （或者由于该函数返回了实际的值，或者由于它抛出了其他类型的异常），那么测试就被视为失败。

检查测试是否失败：

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... FAIL

======================================================================
FAIL: to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=2)
```

太棒了！两个测试都如期地失败了。接着转入被测试的代码并且思考如何才能使得测试通过。

```
def to_roman(n):
    '''convert integer to Roman numeral'''

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

1. 这是Python优雅的快捷方法：一次性的多比较。它等价于 `if not ((0 < n) and (n < 4000))`，但前者更适合阅读。这一行代码应该捕获那些超大的、负值的或者为 0 的输入。
2. 当你改变条件的时候，要确保同步更新那些提示错误信息的可读字符串。 `unittest` 框架并不关心这些，但是如果你的代码抛出描述不正确的异常信息的话会使得手工调试代码变得困难。

我本应该给你展示完整的一系列与本章节不相关的例子来说明一次性多比较的快捷方式是有效的，但是我将仅仅运行本测试用例来证明它的有效性。

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.016s

OK
```

还有一件事情……

还有一个把阿拉伯数字转换成罗马数字的 功能性需求 ：处理非整型数字。

```
>>> import roman3

''

'I'
```

1. 喔，糟糕了。
2. 喔，更糟糕了。两个用例都本该抛出异常的。但却返回了假的结果。

测试非整数并不困难。首先，定义一个 `NotIntegerError` 例外。

```
# roman4.py
class OutOfRangeError(ValueError): pass
<mark>class NotIntegerError(ValueError): pass</mark>
```

然后，编写一个检查 `NotIntegerError` 例外的案例。

```
class ToRomanBadInput(unittest.TestCase):
    .
    .
    .
    def test_non_integer(self):
        '''to_roman should fail with non-integer input'''
 <mark>self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)</mark>
```

然后，检查该测试是否可以正确地失败。

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

======================================================================
FAIL: to_roman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest4.py", line 90, in test_non_integer
    self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
<mark>AssertionError: NotIntegerError not raised by to_roman</mark>

----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (failures=1)
```

编修代码，使得该测试可以通过。

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):
        raise OutOfRangeError('number out of range (must be 1..3999)')

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

1. 内建的 `isinstance()` 方法可以检查一个变量是否属于某一类型（或者，技术上的任何派生类型）。
2. 如果参数 `n` 不是 `int`，则抛出新定义的 `NotIntegerError` 异常。

最后，验证修改后的代码的确通过测试。

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

`to_roman()` 方法通过了所有的测试，而且我也想不出别的测试了，因此，下面着手 `from_roman()` 吧！

可喜的对称性

转换罗马数字为阿拉伯数字的实现难度听起来比反向转换要困难。当然，这种想法不无道理。例如，检查数值是否比0大容易，而检查一个字符串是否为有效的罗马数字则要困难些。但是，我们已经构造了一个用于检查罗马数字的规则表，因此规则表的工作可以免了。

现在剩余的工作就是转换字符串了。正如我们将要看到的一样，多亏我们定义的用于单个罗马数字映射至阿拉伯数字的良好的数据结构，`from_roman()` 的实现本质上与 `to_roman()` 一样简单。

不过，测试先行！为了证明其准确性，我们将需要一个对“已知取值”进行的测试。我们的测试套件已经包含了一个已知取值的映射表，那么，我们就重用它。

```
    def test_from_roman_known_values(self):
        '''from_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

这里看到了令人高兴的对称性。`to_roman()` 与 `from_roman()` 函数是互逆的。前者把整型数字转换为特殊格式化的字符串，而后者则把特殊格式化的字符串转换为整型数字。理论上，我们应该可以使一个数字“绕一圈”，即把数字传递给 `to_roman()` 方法，得到一个字符串;然后把该字符串传入 `from_roman()` 方法，得到一个整型数字，并且跟传给to_roman()方法的数字是一样的。

```
n = from_roman(to_roman(n)) for all values of n
```

在本用例中，“全有取值”是说 `从1到3999` 的所有数值，因为这是 `to_roman()` 方法的有效输入范围。为了表达这两个方法之间的对称性，我们可以设计这样的测试用例，它的测试数据集是从 `1到3999之间` （包括1和3999）的所有数值，首先调用 `to_roman()` ，然后调用 `from_roman()` ，最后检查输出是否与原始输入一致。

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

这些测试连失败的机会都没有。因为我们根本还没定义 `from_roman()` 函数，所以它们仅仅会抛出错误的结果。

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
E.E....
======================================================================
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 78, in test_from_roman_known_values
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

======================================================================
ERROR: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 103, in test_roundtrip
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

----------------------------------------------------------------------
Ran 7 tests in 0.019s

FAILED (errors=2)
```

一个简易的留空函数可以解决此问题。

```
# roman5.py
def from_roman(s):
    '''convert Roman numeral to integer'''
```

（嘿，你注意到了么？我定义了一个除了 docstring 之外没有任何东西的方法。这是合法的 Python 代码。事实上，一些程序员喜欢这样做。“不要留空；写点文档！”）

现在测试用力将会失败。

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
F.F....
======================================================================
FAIL: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 79, in test_from_roman_known_values
    self.assertEqual(integer, result)
AssertionError: 1 != None

======================================================================
FAIL: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 104, in test_roundtrip
    self.assertEqual(integer, result)
AssertionError: 1 != None

----------------------------------------------------------------------
Ran 7 tests in 0.002s

FAILED (failures=2)
```

现在是时候编写 `from_roman()` 函数了。

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:

            result += integer
            index += len(numeral)
    return result
```

1. 此处的匹配模式与 `to_roman()` 完全相同。遍历整个罗马数字数据结构 (一个元组的元组)，与前面不同的是不去一个个地搜索最大的整数，而是搜寻 “最大的”罗马数字字符串。

如果不清楚 `from_roman()` 如何工作，在 `while` 结尾处添加一个 `print` 语句：

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
 <mark>print('found', numeral, 'of length', len(numeral), ', adding', integer)</mark>
```

```
>>> import roman5
>>> roman5.from_roman('MCMLXXII')
found M of length 1, adding 1000
found CM of length 2, adding 900
found L of length 1, adding 50
found X of length 1, adding 10
found X of length 1, adding 10
found I of length 1, adding 1
found I of length 1, adding 1
1972
```

重新执行一遍测试。

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.060s

OK
```

这儿有两个令人激动的消息。一个是 `from_roman()` 对于所有有效输入运转正常，至少对于你测试的已知值是这样。第二个好消息是，完备性测试也通过了。与已知值测试的通过一起来看，你有理由相信 `to_roman()` 和 `from_roman()` 对于所有有效输入值工作正常。(尚不能完全相信，理论上存在这种可能性： `to_roman()` 存在错误而导致一些特定输入会产生错误的罗马数字表示，*and* `from_roman()` 也存在相应的错误，把 `to_roman()` 错误产生的这些罗马数字错误地转换为最初的整数。取决于你的应用程序和你的要求，你或许需要考虑这个可能性；如果是这样，编写更全面的测试用例直到解决这个问题。)

更多错误输入

现在 `from_roman()` 对于有效输入能够正常工作了，是揭开最后一个谜底的时候了：使它正常工作于无效输入的情况下。这意味着要找出一个方法检查一个字符串是不是有效的罗马数字。这比中验证有效的数字输入困难，但是你可以使用一个强大的工具：正则表达式。(如果你不熟悉正则表达式，现在是该好好读读正则表达式那一章节的时候了。)

如你在 个案研究：罗马字母s中所见到的，构建罗马数字有几个简单的规则：使用的字母 `M`，`D`，`C`，`L`，`X`，`V` 和 `I` 。让我们回顾一下：

- 有时字符是叠加组合的。 `I` 是 `1`，`II` 是 `2`,而 `III` 是 `3`. `VI` 是 `6` (从字面上理解，“ `5` 和 `1` ”)，`VII` 是 `7`，而 `VIII` 是 `8`。
- 十位的字符 (`I`、 `X`、 `C` 和 `M`) 可以被重复最多三次。对于 `4`，你则需要利用下一个能够被5整除的字符进行减操作得到。你不能把 `4` 表示为 `IIII`，而应该表示为 `IV` (“比 `5` 小 `1` ”)。`40` 则被写作 `XL` (“比 `50` 小 `10` ”)，`41` 表示为 `XLI`，`42` 表示为 `XLII`，`43` 表示为 `XLIII`，`44` 表示为 `XLIV` (“比 `50` 小 `10`，加上 `5` 小 `1` ”)。
- 有时，字符串是……加法的对立面。通过将某些字符串放的其他一些之前，可以从最终

值中相减。例如，对于 `9`，你需要从下一个最高十位字符串中减去一个值：`8` 是 `VIII`，但 `9` 是 `IX`（“比 `10` 小 `1`”），而不是 `VIIII`（由于 `I` 字符不能重复四次）。`90` 是 `XC`，`900` 是 `CM`。
- 表示 5 的字符不能重复。`10` 总是表示为 `X`，而决不能是 `VV`。`100` 总是 `C`，决不能是 `LL`。
- 罗马数字从左向右读，因此字符的顺序非常重要。`DC` 是 `600`；`CD` 则是完全不同的数字（`400`，“比 `500` 小 `100`”）。`CI` 是 `101`；`IC` 甚至不是合法的罗马数字（因为你不能直接从 `100` 减 `1`；你将不得不将它表示为 `XCIX`，“比 `100` 小 `10`，然后比 `10`”小 `1`）。

因此，有用的测试将会确保 `from_roman()` 函数应当在传入太多重复数字时失败。“太多”是多少取决于数字。

```
class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

另一有效测试是检查某些未被重复的模式。例如，`IX` 代表 `9`，但 `IXIX` 绝不会合法。

```
 def test_repeated_pairs(self):
        '''from_roman should fail with repeated pairs of numerals'''
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

第三个测试应当检测数字是否以正确顺序出现，从最高到最低位。例如，`CL` 是 `150`，而 `LC` 永远是非法的，因为代表 `50` 的数字永远不能在 `100` 数字之前出现。该测试包括一个随机的可选项：`I` 在 `M` 之前，`V` 在 `X` 之前，等等。

```
 def test_malformed_antecedents(self):
        '''from_roman should fail with malformed antecedents'''
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

这些测试中的每个都依赖于 `from_roman()` 引发一个新的例外 `InvalidRomanNumeralError`，而该例外尚未定义。

```
# roman6.py
class InvalidRomanNumeralError(ValueError): pass
```

所有的测试都应该是失败的，因为 `from_roman()` 方法还没有任何有效性检查。（如果没有失败，它们在测什么呢？）

```
you@localhost:~/diveintopython3/examples$ python3 romantest6.py
FFF.......
======================================================================
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 113, in test_malformed_antecedents
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
from_roman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 107, in test_repeated_pairs
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_too_many_repeated_numerals (__main__.FromRomanBadInput)
from_roman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 102, in test_too_many_repeated_numerals
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 10 tests in 0.058s

FAILED (failures=3)
```

好！现在，我们要做的所有事情就是添加正则表达式到 `from_roman()` 中以测试有效的罗马数字。

```
roman_numeral_pattern = re.compile('''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    ''', re.VERBOSE)

def from_roman(s):
    '''convert Roman numeral to integer'''
 <mark>if not roman_numeral_pattern.search(s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))</mark>

    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index : index + len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

再运行一遍测试......

```
you@localhost:~/diveintopython3/examples$ python3 romantest7.py
..........
----------------------------------------------------------------------
Ran 10 tests in 0.066s

OK
```

本年度的虎头蛇尾奖颁发给……单词“OK”，在所有测试通过时，它由 `unittest` 模块输出。

Chapter 10 重构

> " After one has played a vast quantity of notes and more notes, it is simplicity that emerges as the crowning reward of art. " — Frédéric Chopin

深入

就算是竭尽了全力编写全面的单元测试，还是会遇到错误。我所说的“错误”是什么意思？错误是尚未写到的测试实例。

```
>>> import roman7

0
```

1. 这就是错误。和其它无效罗马数字的一系列字符一样，空字符串将引发 `InvalidRomanNumeralError` 例外。

在重现该错误后，应该在修复前写出一个导致该失败情形的测试实例，这样才能描述该错误。

```
class FromRomanBadInput(unittest.TestCase):
    .
    .
    .
    def testBlank(self):
        '''from_roman should fail with blank string'''
```

1. 这段代码非常简单。通过传入一个空字符串调用 `from_roman()` ，并确保其引发一个 `InvalidRomanNumeralError` 例外。难的是发现错误；找到了该错误之后对它进行测试是件轻松的工作。

由于代码有错误，且有用于测试该错误的测试实例，该测试实例将会导致失败：

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
FAIL: from_roman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, '')
<mark>AssertionError: InvalidRomanNumeralError not raised by from_roman</mark>

----------------------------------------------------------------------
Ran 11 tests in 0.171s

FAILED (failures=1)
```

现在 可以修复该错误了。

```
def from_roman(s):
    '''convert Roman numeral to integer'''

        raise InvalidRomanNumeralError('Input can not be blank')
    if not re.search(romanNumeralPattern, s):

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

1. 只需两行代码：一行明确地对空字符串进行检查，另一行为 `raise` 语句。
2. 在本书中还尚未提到该内容，因此现在让我们讲讲 字符串格式化 最后一点内容。从 Python 3.1 起，在格式化标示符中使用位置索引时可以忽略数字。也就是说，无需使用格式化标示符 `{0}` 来指向 `format()` 方法的第一个参数，只需简单地使用 `{}` 而 Python 将会填入正确的位置索引。该规则适用于任何数量的参数；第一个 `{}` 代表 `{0}`，第二个 `{}` 代表 `{1}`，以此类推。

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v

from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 11 tests in 0.156s
```

1. 现在空字符串测试实例通过了测试，也就是说错误被修正了。
2. 所有其它测试实例仍然可以通过，说明该错误修正没有破坏其它部分。代码编写结束。

用此方式编写代码将使得错误修正变得更困难。简单的错误（像这个）需要简单的测试实例；复杂的错误将会需要复杂的测试实例。在以测试为中心的环境中，由于必须在代码中精确地描述错误（编写测试实例），然后修正错误本身，看起来 好像 修正错误需要更多的时间。而如果测试实例无法正确地通过，则又需要找出到底是修正方案有错误，还数测试实例本身就有错误。然而从长远看，这种在测试代码和经测试代码之间的来回折腾是值得的，因为这样才更有可能在第一时间修正错误。同时，由于可以对新代码轻松地重新运行 所有 测试实例，在修正新代码时破坏旧代码的机会更低。今天的单元测试就是明天的回归测试。

控制需求变化

为了获取准确的需求，尽管已经竭力将客户“钉”在原地，并经历了反复剪切、粘贴的痛苦，但需求仍然会变化。大多数客户在看到产品之前不知道自己想要什么，而且就算知道，他们也不擅长清晰地表述自己的想法。而即便擅长表述，他们在下一个版本中也会提出更多要求。因此，必须随时准备好更新测试实例以应对需求变化。

举个例子来说，假定我们要扩展罗马数字转换函数的能力范围。正常情况下，罗马数字中的任何一个字符在同一行中不得重复出现三次以上。但罗马人却愿意该规则有个例外：通过一行中的 4 个 `M` 字符来代表 `4000` 。进行该修改后，将会把可转换数字的范围从 `1..3999` 拓展为 `1..4999` 。但首先必须对测试实例进行一些修改。

```
class KnownValues(unittest.TestCase):
    known_values = ( (1, 'I'),
                     .
                     .
                     .
                   (3999, 'MMMCMXCIX'),

                   (4500, 'MMMMD'),
                   (4888, 'MMMMDCCCLXXXVIII'),
                   (4999, 'MMMMCMXCIX') )

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''

    .
    .
    .

class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''

            self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, s)

    .
    .
    .

class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''

            numeral = roman8.to_roman(integer)
            result = roman8.from_roman(numeral)
            self.assertEqual(integer, result)
```

1. 现有的已知数值不会变（它们依然是合理的测试数值），但必须在 `4000` 范围之内（外）增加一些。在此，我已经添加了 `4000` (最短)、 `4500` (第二短)、 `4888` (最长) 和 `4999` (最大)。
2. “过大值输入” 的定义已经发生了变化。该测试用于通过传入 `4000` 调用 `to_roman()` 并期望引发一个错误；目前 `4000-4999` 是有效的值，必须将该值调整为 `5000` 。
3. “太多重复数字”的定义也发生了变化。该测试通过传入 `'MMMM'` 调用 `from_roman()` 并预期发生一个错误；目前 `MMMM` 被认定为有效的罗马数字，必须将该条件修改为 `'MMMMM'` 。
4. 对范围内的每个数字进行完整循环测试，从 `1` 到 `3999` 。由于范围已经进行了拓展，该 `for` 循环同样需要修改为以 `4999` 为上限。

现在，测试实例已经按照新的需求进行了更新，但代码还没有，因按照预期，某些测试实例将返回失败结果。

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok

to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
ERROR: from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 82, in test_from_roman_known_values
    result = roman9.from_roman(numeral)
  File "C:\home\diveintopython3\examples\roman9.py", line 60, in from_roman
    raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
<mark>roman9.InvalidRomanNumeralError: Invalid Roman numeral: MMMM</mark>

======================================================================
ERROR: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 76, in test_to_roman_known_values
    result = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
<mark>roman9.OutOfRangeError: number out of range (must be 0..3999)</mark>

======================================================================
ERROR: from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 131, in testSanity
    numeral = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
<mark>roman9.OutOfRangeError: number out of range (must be 0..3999)</mark>

----------------------------------------------------------------------
Ran 12 tests in 0.171s

FAILED (errors=3)
```

1. 一旦遇到 `'MMMM'` ， `from_roman()` 已知值测试将会失败，因为 `from_roman()` 仍将其视为无效罗马数字。
2. 一旦遇到 `4000` ， `to_roman()` 已知值测试将会失败，因为 `to_roman()` 仍将其视为超范围数字。
3. 而往返（译注：指在普通数字和罗马数字之间来回转换）检查遇到 `4000` 时也会失败，因为 `to_roman()` 仍认为其超范围。

现在，我们有了一些由新需求导致失败的测试实例，可以考虑修正代码让它与新测试实例一致起来。（刚开始编写单元测试的时候，被测试代码绝不会在测试实例“之前”出现确实让人感觉有点怪。）尽管编码工作被置后安排，但还是不少要做的事情，一旦与测试实例相符，编码工作就可以结束了。一旦习惯单元测试后，您可能会对自己曾在编程时不进行测试感到很奇怪。）

```
roman_numeral_pattern = re.compile('''
    ^                   # beginning of string

    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    ''', re.VERBOSE)

def to_roman(n):
    '''convert integer to Roman numeral'''

        raise OutOfRangeError('number out of range (must be 1..4999)')
    if not isinstance(n, int):
        raise NotIntegerError('non-integers can not be converted')

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def from_roman(s):
    .
    .
    .
```

1. 根本无需对 `from_roman()` 函数进行任何修改。唯一需要修改的是 `roman_numeral_pattern` 。仔细观察下，将会发现我已经在正则表达式的第一部分中将 `M` 字符的数量从 `3` 优化为 `4` 。该修改将允许等价于 `4999` 而不是 `3999` 的罗马数字。实际的 `from_roman()` 函数完全是通用的；它只查找重复的罗马数字字符并将它们加起来，而不关心它们重复了多少次。之前无法处理 `'MMMM'` 的唯一原因是我们通过正则表达式匹配明确地阻止了它这么做。
2. `to_roman()` 函数只需在范围检查中进行一个小改动。将之前检查 `0 < n < 4000` 的地方现在修改为检查 `0 < n < 5000` 。同时修改 `引发` 的错误信息，以体现新的可接受范围（`1..4999` 取代 `1..3999` ）。无需对函数剩下部分进行任何修改；它已经能够应对新的实例。（它将对找到的每个千位增加 `'M'` ；如果给定 `4000` ，它将给出 `'MMMM'` 。之前它不这么做的唯一原因是我们通过范围检查明确地阻止了它。）

所需做的就是这两处小修改，但你可能会有点怀疑。嗨，别光听我说，你自己看看吧。

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.203s
```

1. 所有测试实例均通过了。代码编写结束。

全面单元测试的意思是：无需依赖某个程序员来说“相信我吧。”

重构

关于全面单元测试，最美妙的事情不是在所有的测试实例通过后的那份心情，也不是别人抱怨你破坏了代码，而你通过实践 证明 自己没有时的快感。单元测试最美妙之处在于它给了你大刀阔斧进行重构的自由。

重构是修改可运作代码，使其表现更佳的过程。通常，“更佳”指的是“更快”，但它也可能指的是“占用更少内存“、”占用更少磁盘空间“或者”更加简洁”。对于你的环境、你的项目来说，无论重构意味着什么，它对程序的长期健康都至关重要。

本例中，“更佳”的意思既包括“更快”也包括“更易于维护”。具体而言，因为用于验证罗马数字的正则表达式生涩冗长，该 `from_roman()` 函数比我所希望的更慢，也更加复杂。现在，你可能会想，“当然，正则表达式就又臭又长的，难道我有其它办法验证任意字符串是否为罗马数字吗？”

答案是：只针对 5000 个数进行转换；为什么不知建立一个查询表呢？意识到 根本不需要使用正则表达式 之后，这个主意甚至变得更加理想了。在建立将整数转换为罗马数字的查询表的同时，还可以建立将罗马数字转换为整数的逆向查询表。在需要检查任意字符串是否是有效罗马数字的时候，你将收集到所有有效的罗马数字。“验证”工作简化为一个简单的字典查询。

最棒的是，你已经有了一整套单元测试。可以修改模块中一半以上的代码，而单元测试将会保持不变。这意味着可以向你和其他人证明：新代码运作和最初的一样好。

```
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
class InvalidRomanNumeralError(ValueError): pass

roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),
                     ('I',  1))

to_roman_table = [ None ]
from_roman_table = {}

def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]

def build_lookup_tables():
    def to_roman(n):
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)
        to_roman_table.append(roman_numeral)
        from_roman_table[roman_numeral] = integer

build_lookup_tables()
```

让我们打断一下，进行一些剖析工作。可以说，最重要的是最后一行：

```
build_lookup_tables()
```

可以注意到这是一次函数调用，但没有 `if` 语句包裹住它。这不是 `if __name__ == '__main__'` 语块；模块被导入时 它将会被调用。（重要的是必须明白：模块将只被导入一次，随后被缓存了。如果导入一个已导入模块，将不会导致任何事情发生。因此这段代码将只在第一此导入时运行。）

那么，该 `build_lookup_tables()` 函数究竟进行了哪些操作呢?很高兴你问这个问题。

```
to_roman_table = [ None ]
from_roman_table = {}
.
.
.
def build_lookup_tables():

        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):

        from_roman_table[roman_numeral] = integer
```

1. 这是一段聪明的程序代码……也许过于聪明了。上面定义了 `to_roman()` 函数；它在查询表中查找值并返回结果。而 `build_lookup_tables()` 函数重定义了 `to_roman()` 函数用于实际操作（像添加查询表之前的例子一样）。在 `build_lookup_tables()` 函数内部，对 `to_roman()` 的调用将会针对该重定义的版本。一旦 `build_lookup_tables()` 函数退出，重定义的版本将会消失 — 它的定义只在 `build_lookup_tables()` 函数的作用域内生效。
2. 该行代码将调用重定义的 `to_roman()` 函数，该函数实际计算罗马数字。
3. 一旦获得结果（从重定义的 `to_roman()` 函数），可将整数及其对应的罗马数字添加到两个查询表中。

查询表建好后，剩下的代码既容易又快捷。

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
```

1. 像前面那样进行同样的边界检查之后，`to_roman()` 函数只需在查询表中查找并返回适当

的值。

2. 同样，`from_roman()` 函数也缩水为一些边界检查和一行代码。不再有正则表达式。不再有循环。O(1) 转换为或转换到罗马数字。

但这段代码可以运作吗？为什么可以，是的它可以。而且我可以证明。

```
you@localhost:~/diveintopython3/examples$ python3 romantest10.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------

OK
```

1. 它不仅能够回答你的问题，还运行得非常快！好象速度提升了 10 倍。当然，这种比较并不公平，因为此版本在导入时耗时更长（在建造查询表时）。但由于只进行一次导入，启动的成本可以由对 `to_roman()` 和 `from_roman()` 函数的所有调用摊薄。由于该测试进行几千次函数调用（来回单独测试上万次），节省出来的效率成本得以迅速提升！

这个故事的寓意是什么？

- 简单是一种美德。
- 特别在涉及到正则表达式的时候。
- 单元测试令你在进行大规模重构时充满自信。

摘要

单元测试是一个威力强大的概念，如果正确实施，不但可以降低维护成本，还可以提高长期项目的灵活性。但同时还必须明白：单元测试既不是灵丹妙药，也不是解决问题的魔术，更不是银弹。编写良好的测试实例非常艰难，确保它们时刻保持最新必须成为一项纪律（特别在客户要求关键错误修正时）。单元测试不是功能测试、集成测试或用户承受能力测试等其它测试的替代品。但它是可行的、行之有效的，见识过其功用后，你将对之前曾没有用它而感到奇怪。

这几章覆盖的内容很多，很大一部分都不是 Python 所特有的。许多语言都有单元测试框架，但所有框架都要求掌握同一基本概念：

- 设计测试实例是件具体、自动且独立的工作。
- 在编写被测试代码 之前 编写测试实例。

- 编写用于检查好输入并验证正确结果的测试
- 编写用于测试"坏"输入并做出正确失败响应的测试。
- 编写并更新测试实例以反映新的需求
- 毫不留情地重构以提升性能、可扩展性、可读性、可维护性及任何缺乏的特性。

Chapter 11 文件

> " A nine mile walk is no joke, especially in the rain. " — Harry Kemelman, The Nine Mile Walk

概要

在没有安装任何一个应用程序之前，我的笔记本上Windows系统有38,493个文件。安装Python 3后，大约增加了3,000个文件。文件是每一个主流操作系统的主要存储模型；这种观念如此根深蒂固以至于难以想出一种替代物。打个比方，你的电脑实际上就是泡在文件里了。

读取文本文件

在读取文件之前，你需要先打开它。在Python里打开一个文件很简单：

```
a_file = open('examples/chinese.txt', encoding='utf-8')
```

Python有一个内置函数 `open()` ，它使用一个文件名作为其参数。在以上代码中，文件名是 `'examples/chinese.txt'` 。关于这个文件名，有五件值得一讲的事情：

1. 它不仅是一个文件的名字；实际上，它是文件路径和文件名的组合；一般来说，文件打开函数应该有两个参数 — 路径和文件名 — 但是函数 `open()` 只使用一个参数。在Python里，当你使用“filename,”作为参数的时候，你可以将部分或者全部的路径也包括进去。
2. 在这个例子中，目录路径中使用的是斜杠(forward slash)，但是我并没有说明我正在使用的操作系统。Windows使用反斜杠来表示子目录，但是Mac OS X和Linux使用斜杠。但是，在Python中，斜杠永远都是正确的，即使是在Windows环境下。
3. 不使用斜杠或者反斜杠的路径被称作相对路径*(relative path)*。你也许会问，相对于什么呢？耐心一些，伙计。
4. “filename,”参数是一个字符串。所有现代的操作系统（甚至Windows！）使用Unicode编码方式来存储文件名和目录名。Python 3全面支持非ASCII编码的路径。
5. 文件不一定需要在本地磁盘上。也许你挂载了一个网络驱动器。它也可以是一个完全虚拟的文件系统(an entirely virtual filesystem)上的文件。只要你的操作系统认为它是一个文件，并且能够以文件的方式访问，那么，Python就能打开它。

但是对 `open()` 函数的调用不局限于 `filename` 。还有另外一个叫做 `encoding` 参数。天哪，似乎非常耳熟的样子！

字符编码抬起了它腌臜的头...

字节即字节；字符是一种抽象。字符串由使用Unicode编码的字符序列构成。但是磁盘上的文件不是Unicode编码的字符序列。文件是字节序列。所以你可能会想，如果从磁盘上读取一个“文本文件”，Python是怎样把那个字节序列转化为字符序列的呢？实际上，它是根据特定的字符解码算法来解释这些字节序列，然后返回一串使用Unicode编码的字符（或者也称为字符串）。

```
# This example was created on Windows. Other platforms may
# behave differently, for reasons outlined below.
# 这个样例在Windows平台上创建。其他平台可能会有不同的表现，理由描述在下边
>>> file = open('examples/chinese.txt')
>>> a_string = file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python31\lib\encodings\cp1252.py", line 23, in decode
    return codecs.charmap_decode(input,self.errors,decoding_table)[0]
UnicodeDecodeError: 'charmap' codec can't decode byte 0x8f in position 28: character maps
>>>
```

默认的编码方式是平台相关的。

刚才发生了什么？由于你没有指定字符编码的方式，所以Python被迫使用默认的编码。那么默认的编码方式是什么呢？如果你仔细看了跟踪信息(traceback)，错误出现在 `cp1252.py` ，这意味着Python此时正在使用CP-1252作为默认的编码方式。（在运行微软视窗操作系统的机器上，CP-1252是一种常用的编码方式。）CP-1252的字符集不支持这个文件上的字符编码，所以它以这个可恶的 `UnicodeDecodeError` 错误读取失败。

但是，还有更糟糕的！因为默认的编码方式是平台相关的*(platform-dependent)*，所以，当前的代码也许能够在你的电脑上运行（如果你的机器的默认编码方式是UTF-8），但是当你把这份代码分发给其他人的时候可能就会失败（因为他们的默认编码方式可能跟你的不一样，比如说CP-1252）。

> ☞如果你需要获得默认编码的信息，则导入 `locale` 模块，然后调用 `locale.getpreferredencoding()` 。在我安装了Windows的笔记本上，它的返回值是 `'cp1252'` ，但是在我楼上安装了Linux的台式机上边，它返回 `'UTF8'` 。你看，即使在我自己家里我都不能保证一致性(consistency)！你的运行结果也许不一样（即使在Windows平台上），这依赖于操作系统的版本和区域/语言选项的设置。这就是为什么每次打开一个文件的时候指定编码方式是如此重要了。

流对象

到目前为止，我们都知道Python有一个内置的函数叫做 `open()` 。 `open()` 函数返回一个流对象（stream object），它拥有一些用来获取信息和操作字符流的方法和属性。

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')

'examples/chinese.txt'

'utf-8'

'r'
```

1. `name` 属性反映的是当你打开文件时传递给 `open()` 函数的文件名。它没有被标准化(normalize)成绝对路径。
2. 同样的， `encoding` 属性反映的是在你调用 `open()` 函数时指定的编码方式。如果你在打开文件的时候没有指定编码方式（不好的开发人员！），那么 `encoding` 属性反映的是 `locale.getpreferredencoding()` 的返回值。
3. `mode` 属性会告诉你被打开文件的访问模式。你可以传递一个可选的 `mode` 参数给 `open()` 函数。如果在打开文件的时候没有指定访问模式，Python默认设置模式为 `'r'`， 意思是“在文本模式下以只读的方式打开。”在这章的后面你会看到，文件的访问模式有各种用途；不同模式能够使你写入一个文件，追加到一个文件，或者以二进制模式打开一个文件（在这种情况下，你处理的是字节，不再是字符）。

> ☞ `open()` 函数的文档列出了所有可用的文件访问模式。

从文本文件读取数据

在打开文件以后，你可能想要从某处开始读取它。

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')

'Dive Into Python 是为有经验的程序员编写的一本 Python 书。\n'

''
```

1. 只要成功打开了一个文件（并且指定了正确的编码方式），你只需要调用流对象的 `read()` 方法即可以读取它。返回的结果是文件的一个字符串表示。
2. 也许你会感到意外，再次读取文件不会产生一个异常。Python不认为到达了文件末尾(end-of-file)还继续执行读取操作是一个错误；这种情况下，它只是简单地返回一个空字符串。

无论何时，打开文件时指定 `encoding` 参数。

如果想要重新读取文件呢？

```
# continued from the previous example
# 接着前一个例子

''

0

'Dive Into Python'

' '
>>> a_file.read(1)
'是'

20
```

1. 由于你依旧在文件的末尾，继续调用 `read()` 方法只会返回一个空字符串。
2. `seek()` 方法使定位到文件中的特定字节。
3. `read()` 方法可以使用一个可选的参数，即所要读取的字符个数。
4. 只要愿意，你甚至可以一次读取一个字符。
5. 16 + 1 + 1 = … 20?

我们再来做一遍。

```
# continued from the previous example
# 继续上一示例

17

'是'

20
```

1. 移动到第17th个字节位置。
2. 读取一个字符。
3. 当前在第20个字节位置处。

你是否已经注意到了？ `seek()` 和 `tell()` 方法总是以字节的方式计数，但是，由于你是以文本文件的方式打开的， `read()` 方法以字符的个数计数。中文字符的UTF-8编码需要多个字节。而文件里的英文字符每一个只需要一个字节来存储，所以你可能会产生这样的误解： `seek()` 和 `read()` 方法对相同的目标计数。而实际上，只有对部分字符的情况是这样的。

但是，还有更糟的！

```

18

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a_file.read(1)
  File "C:\Python31\lib\codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0x98 in position 0: unexpected code by
```

1. 定位到第18th个字节，然后试图读取一个字符。
2. 为什么这里会失败？因为在第18个字节处不存在字符。距离此处最近的字符从第17个字节开始（长度为三个字节）。试图从一个字符的中间位置读取会导致程序以 `UnicodeDecodeError` 错误失败。

关闭文件

打开文件会占用系统资源，根据文件的打开模式不同，其他的程序也许不能够访问它们。当已经完成了对文件的操作后就立即关闭它们，这很重要。

```
# continued from the previous example
# 继续前面的例子
>>> a_file.close()
```

然而，这还不够(anticlimactic)。

流对象 `a_file` 仍然存在；调用 `close()` 方法并没有把对象本身销毁。所以这并不是非常有效。

```
# continued from the previous example
# 接着上一示例

Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    a_file.read()
ValueError: I/O operation on closed file.

Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    a_file.seek(0)
ValueError: I/O operation on closed file.

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    a_file.tell()
ValueError: I/O operation on closed file.

True
```

1. 不能读取已经关闭了的文件；那样会引发一个 `IOError` 异常。
2. 也不能对一个已经关闭了的文件执行定位操作。
3. 由于文件已经关闭了，所以也就不存在所谓当前的位置了，所以 `tell()` 也会失败。
4. 也许你会有些意外，文件已经关闭，调用原来流对象的 `close()` 方法并没有引发异常。其实那只是一个空操作(no-op)而已。
5. 已经关闭了的流对象确实还有一个有用的属性：`closed` 用来确认文件是否已经被关闭了。

自动关闭文件

`try..finally` 也行。但是 `with` 更好

流对象有一个显式的 `close()` 方法，但是如果代码有缺陷，在调用 `close()` 方法以前就崩溃了呢？理论上，那个文件会在相当长的一段时间内一直打开着，这是没有必要地。当你在自己的机器上调试的时候，这不算什么大问题。但是当这种代码被移植到服务器上运行，也许就得三思了。

对于这种情况，Python 2有一种解决办法： `try..finally` 块。这种方法在Python 3里仍然有效，也许你可以在其他人的代码，或者从比较老的被移植到Python 3的代码中看到它。但是Python 2.5引入了一种更加简洁的解决方案，并且Python 3将它作为首选方案： `with` 语句。

```
with open('examples/chinese.txt', encoding='utf-8') as a_file:
    a_file.seek(17)
    a_character = a_file.read(1)
    print(a_character)
```

这段代码调用了 `open()` 函数，但是它却一直没有调用 `a_file.close()` 。 `with` 语句引出一个代码块，就像 `if` 语句或者 `for` 循环一样。在这个代码块里，你可以使用变量 `a_file` 作为 `open()` 函数返回的流对象的引用。所以流对象的常规方法都是可用的 — `seek()` ， `read()` ，无论你想要调用什么。当 `with` 块结束时，*Python*自动调用 `a_file.close()` 。

这就是它与众不同的地方：无论你以何种方式跳出 `with` 块，Python会自动关闭那个文件…即使是因为未处理的异常而“exit”。是的，即使代码中引发了一个异常，整个程序突然中止了，Python也能够保证那个文件能被关闭掉。

> ☞从技术上说， `with` 语句创建了一个运行时环境(runtime context)。在这几个样例中，流对象的行为就像一个上下文管理器(context manager)。Python创建了 `a_file` ，并且告诉它正在进入一个运行时环境。当 `with` 块结束的时候，Python告诉流对象它正在退出这个运行时环境，然后流对象就会调用它的 `close()` 方法。请阅读 附录B，“能够在 `with` 块中使用的类”以获取更多细节。

`with` 语句不只是针对文件而言的；它是一个用来创建运行时环境的通用框架(generic framework)，告诉对象它们正在进入和离开一个运行时环境。如果该对象是流对象，那么它就会做一些类似文件对象一样有用的动作（就像自动关闭文件！）。但是那个行为是被流对象自身定义的，而不是在 `with` 语句中。还有许多跟文件无关的使用上下文管理器(context manager)的方法。在这章的后面可以看到，你甚至可以自己创建它们。

一次读取一行数据

正如你所想的，一行数据就是这样 — 输入一些单词，按 `ENTER` 键，然后就在新的一行了。一行文本就是一串被某种东西分隔的字符，到底是被什么分隔的呢？好吧，这有些复杂，因为文本文件可以使用几个不同的字符来标记行末(end of a line)。每种操作系统都有自己的规矩。有一些使用回车符(carriage return)，另外一些使用换行符(line feed)，还有一些在行末同时使用这两个字符来标记。

其实你可以舒口气了，因为*Python*默认会自动处理行的结束符。如果你告诉它，“我想从这个文本文件一次读取一行，”Python自己会弄明白这个文本文件到底使用哪种方式标记新行，然后正确工作。

> ☞如果想要细粒度地控制(fine-grained control)使用哪种新行标记符，你可以传递一个可选的参数 `newline` 给 `open()` 函数。请阅读 `open()` 函数的文档以获取更多细节。

那么，实际中你会怎样做呢？我是指一次读取文件的一行。它如此简单优美...

```
line_number = 0
        line_number += 1
```

1. 使用 `with` 语句，安全地打开这个文件，然后让Python为你关闭它。
2. 为了一次读取文件的一行，使用 `for` 循环。是的，除了像 `read()` 这样显式的方法，流对象也是一个迭代器(*iterator*)，它能在你每次请求一个值时分离出单独的一行。
3. 使用字符串的 `format()` 方法，你可以打印出行号和行自身。格式说明符 `{:>4}` 的意思是“使用最多四个空格使之右对齐，然后打印此参数。”变量 `a_line` 是包括回车符等在内的完整的一行。字符串方法 `rstrip()` 可以去掉尾随的空白符，包括回车符。

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
 1 Dora
   2 Ethan
   3 Wesley
   4 John
   5 Anne
   6 Mike
   7 Chris
   8 Sarah
   9 Alex
  10 Lizzie
```

> 是否遇到了这个错误？
>
> ```
> you@localhost:~/diveintopython3$ python3 examples/oneline.py
> Traceback (most recent call last):
>   File "examples/oneline.py", line 4, in <module>
>     print('{:>4} {}'.format(line_number, a_line.rstrip()))
> ValueError: zero length field name in format
> ```
>
> 如果结果是这样，也许你正在使用Python 3.0。你真的应该升级到Python 3.1。
>
> Python 3.0支持字符串格式化，但是只支持显式编号了的格式说明符。Python 3.1允许你在格式说明符里省略参数索引号。作为比照，下面是一个Python 3.0兼容的版本。
>
> ```
> print('{<mark>0</mark>:>4} {<mark>1</mark>}'.format(line_
> ```

写入文本文件

打开文件然后开始写入即可。

写入文件的方式和从它们那儿读取很相似。首先打开一个文件，获取流对象，然后你调用一些方法作用在流对象上来写入数据到文件，最后关闭文件。

为了写入而打开一个文件，可以使用 `open()` 函数，并且指定写入模式。有两种文件模式用于写入：

- “写”模式会重写文件。传递 `mode='w'` 参数给 `open()` 函数。
- “追加”模式会在文件末尾添加数据。传递 `mode='a'` 参数给 `open()` 函数。

如果文件不存在，两种模式下都会自动创建新文件，所以就不需要“如果文件还不存在，创建一个新的空白文件以能够打开它”这种琐碎的过程了。所以，只需要打开一个文件，然后开始写入即可。

在完成写入后你应该马上关闭文件，释放文件句柄(file handle)，并且保证数据被完整地写入到了磁盘。跟读取文件一样，可以调用流对象的 `close()` 方法，或者你也可以使用 `with` 语句让Python为你关闭文件。我敢打赌，你肯定能猜到我推荐哪种方案。

```
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
test succeeded

...     a_file.write('and again')
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
```

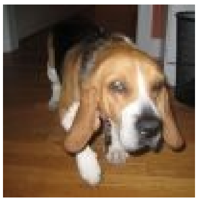
1. 大胆地创建新文件 `test.log` （或者重写已经存在的文件），然后以写入方式打开文件。参数 `mode='w'` 的意思是文件以写入的模式打开。是的，这听起来似乎比较危险。我希望你确定不再关心那个文件以前的内容（如果有的话），因为那份数据已经没了。
2. 你可以通过 `open()` 函数返回的流对象的 `write()` 方法来给新打开的文件添加数据。当 `with` 块结束的时候，Python自动关闭文件。
3. 多么有趣，我们再试一次。这一次，使用 `with='a'` 参数来添加数据到文件末尾，而不是重写它。追加模式绝不会破坏现有文件的内容。
4. 原来写入的行，还有追加上去的第二行现在都在文件 `test.log` 里了。同时请注意，回车符没有被包括进去。你可以通过 `'\n'` 写入一个回车符。由于一开始没有这样做，所有写入到文件的数据现在都在同一行。

再次讨论字符编码

你是否注意到当你在打开文件用于写入数据的时候传递给 `open()` 函数的 `encoding` 参数。它“非常重要”，不要忽略了！就如你在这章开头看到的，文件中并不存在字符串，它们由字节组成。只有当你告诉Python使用何种编码方式把字节流转换为字符串，从文件读取“字符串”才

成为可能。相反地，写入文本到文件面临同样的问题。实际上你不能直接把字符写入到文件；字符只是一种抽象。为了写入字符到文件，Python需要知道如何将字符串转换为字节序列。唯一能保证正确地执行转换的方法就是当你为写入而打开一个文件的时候，指定 `encoding` 参数。

二进制文件



不是所有的文件都包含文本内容。有一些还包含了我可爱的狗的照片。

```
'rb'

'examples/beauregard.jpg'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.BufferedReader' object has no attribute 'encoding'
```

1. 用二进制模式打开文件很简单，但是很精细。与文本模式唯一不同的是 `mode` 参数包含一个字符 `'b'` 。
2. 以二进制模式打开文件得到的流对象与之前的有很多相同的属性，包括 `mode` 属性，它记录了你调用 `open()` 函数时指定的 `mode` 参数的值。
3. 二进制文件的流对象也有 `name` 属性，就如文本文件的流对象一样。
4. 然而，确实有不同之处：二进制的流对象没有 `encoding` 属性。你能明白其中的道理的，对吧？现在你读写的是字节，而不是字符串，所以Python不需要做转换工作。从二进制文件里读出的跟你所写入的是完全一样的，所以没有执行转换的必要。

我是否提到当前正在读取字节？噢，的确如此。

```
# continued from the previous example
# 继续前一样例
>>> an_image.tell()
0

>>> data
b'\xff\xd8\xff'

<class 'bytes'>

3
>>> an_image.seek(0)
0
>>> data = an_image.read()
>>> len(data)
3150
```

1. 跟读取文本文件一样，你也可以从二进制文件一次读一点儿。但是它们之间有一个重大的不同之处处&#hellip;
2. >&#hellip;你正在读取字节，而不是字符串。由于你以二进制模式打开文件，`read()` 方法每次读取指定的字节数，而非字符数。
3. 这就意味着，你传递给 `read()` 方法的数目和你从 `tell()` 方法得到的位置序号不会出现意料之外的不匹配(unexpected mismatch)

非文件来源的流对象

使用 `read()` 方法即可从虚拟文件读取数据。

想象一下你正在编写一个库(library)，其中有一库函数用来从文件读取数据。它使用文件名作为参数，以只读的方式打开文件，读取数据，关闭文件，返回。但是你不应该只做到这个程度。你的API应该能够接纳任意的类型的流对象。

最简单的情况，只要对象包含 `read()` 方法，这个方法使用一个可选参数 `size` 并且返回值为一个串，它就是是流对象。不使用 `size` 参数调用 `read()` 的时候，这个方法应该从输入源读取所有可读的信息然后以单独的一个值返回所有数据。当使用 `size` 参数调用 `read()` 时，它从输入源读取并返回指定量的数据。当再一次被调用时，它从上一次离开的地方开始读取并返回下一个数据块。

这听起来跟你从打开一个真实文件得到的流对象一样。不同之处在于你不再受限于真实的文件。能够“读取”的输入源可以是任何东西：网页，内存中的字符串，甚至是另外一个程序的输出。只要你的函数使用的是流对象，调用对象的 `read()` 方法，你可以处理任何行为与文件类似的输入源，而不需要为每种类型的输入指定特别的代码。

```
>>> a_string = 'PapayaWhip is the new black.'

'PapayaWhip is the new black.'

''

0

'PapayaWhip'
>>> a_file.tell()
10
>>> a_file.seek(18)
18
>>> a_file.read()
'new black.'
```

1. `io` 模块定义了 `StringIO` 类，你可以使用它来把内存中的字符串当作文件来处理。
2. 为了从字符串创建一个流对象，可以把想要作为“文件”使用的字符串传递给 `io.StringIO()` 来创建一个 `StringIO` 的实例。
3. 调用 `read()` 方法“读取”整个“文件”，以 `StringIO` 对象为例即返回原字符串。
4. 就像一个真实的文件一样，再次调用 `read()` 方法返回一个空串。

5. 通过使用 `StringIO` 对象的 `seek()` 方法，你可以显式地定位到字符串的开头，就像在一个真实的文件中定位一样。
6. 通过传递 `size` 参数给 `read()` 方法，你也可以以数据块的形式读取字符串。

> ☞ `io.StringIO` 让你能够将一个字符串作为文本文件来看待。另外还有一个 `io.ByteIO` 类，它允许你将字节数组当做二进制文件来处理。

处理压缩文件

Python标准库包含支持读写压缩文件的模块。有许多种不同的压缩方案；其中，gzip和bzip2是非Windows操作系统下最流行的两种压缩方式。

`gzip` 模块允许你创建用来读写gzip压缩文件的流对象。该流对象支持 `read()` 方法（如果你以读取模式打开）或者 `write()` 方法（如果你以写入模式打开）。这就意味着，你可以使用从普通文件那儿学到的技术来直接读写*gzip*压缩文件，而不需要创建临时文件来保存解压缩了的数据。

作为额外的功能，它也支持 `with` 语句，所以当你完成了对gzip压缩文件的操作，Python可以为你自动关闭它。

```
you@localhost:~$ python3
>>> import gzip
...   z_file.write('A nine mile walk is no joke, especially in the rain.'.encode('utf-8'))
...
>>> exit()
-rw-r--r--  1 mark mark    79 2009-07-19 14:29 out.log.gz
A nine mile walk is no joke, especially in the rain.
```

1. 你应该问题以二进制模式打开gzip压缩文件。（注意 `mode` 参数里的 `'b'` 字符。）
2. 我在Linux系统上完成的这个例子。如果你对命令行不熟悉，这条命令用来显示刚才你在Python shell创建的gzip压缩文件的“长清单(long listings)”，你可以看到，它有79个字节长。而实际上这个值比一开始的字符串还要长！由于gzip文件包括了一个固定长度的文件头来存放一些关于文件的元数据(metadata)，所以它对于极小的文件来说效率不高。
3. `gunzip` 命令（发音：“gee-unzip”）解压缩文件然后保存其内容到一个与原来压缩文件同名的新文件中，并去掉其 `.gz` 扩展名。
4. `cat` 命令显示文件的内容。当前文件包含了原来你从Python shell直接写入到压缩文件 `out.log.gz` 的那个字符串。

标准输入、输出和错误

`sys.stdin`, `sys.stdout`, `sys.stderr`.

命令行高手已经对标准输入，标准输出和标准错误的概念相当熟悉了。这部分内容是对另一部分还不熟悉的人员准备的。

标准输出和标准错误（通常缩写为 `stdout` 和 `stderr` ）是被集成到每一个类UNIX操作系统中的两个管道(pipe)，包括Mac OS X和Linux。当你调用 `print()` 的时候，需要打印的内容即被发送到 `stdout` 管道。当你的程序出错并且需要打印跟踪信息(traceback)时，它们被发送到 `stderr` 管道。默认地，这两个管道都被连接到你正在工作的终端窗口上(terminal window)；当你的程序打印某些东西，你可以在终端上看到这些输出，当程序出错，你也可以从终端上看到这些错误信息。在图形化的Python shell里， `stdout` 和 `stderr` 管道默认连接到“交互式窗口(Interactive Window)”

```
>>> for i in range(3):

PapayaWhip
PapayaWhip
PapayaWhip
>>> import sys
>>> for i in range(3):

is theis theis the
>>> for i in range(3):

new blacknew blacknew black
```

1. 循环调用 `print()` 函数。没有什么特别的。
2. `stdout` 被定义在 `sys` 模块里，它是一个流对象(stream object)。使用任意字符串调用其 `write()` 函数会按原样输出。事实上，这就是 `print()` 函数实际在做的事情；它在串的结尾添加一个回车符，然后调用 `sys.stdout.write` 。
3. 最简单的情况下， `sys.stdout` 和 `sys.stderr` 把他们的输出发送到同一个位置：Python IDE（如果你在那里执行操作），或者终端（如果你从命令行执行Python指令）。跟标准输出一样，标准错误也不会自动为你添加回车符。如果你需要回车符，你需要手工写入回车符到标准错误。

`sys.stdout` 和 `sys.stderr` 都是流对象，但是他们都只支持写入。试图调用他们的 `read()` 方法会引发 `IOError` 异常。

```
>>> import sys
>>> sys.stdout.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: not readable
```

标准输出重定向

`sys.stdout` 和 `sys.stderr` 都是流对象，尽管他们只支持写入。但是他们是变量而不是常量。这就意味着你可以给它们赋上新值 — 任意其他流对象 — 来重定向他们的输出。

```
import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):
        sys.stdout = self.out_old

print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

验证一下：

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/examples$ cat out.log
B
```

> 你是否遇到了以下错误？
>
> ```
> you@localhost:~/diveintopython3/examples$ python3 stdout.py
>  File "stdout.py", line 15
>     with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_f
>                                                                ^
> SyntaxError: invalid syntax
> ```
>
> 如果是这样，你可能正在使用Python 3.0。应该升级到Python 3.1。
>
> Python 3.0支持 `with` 语句，但是每个语句只能使用一个上下文管理器。Python 3.1允许你在一条 `with` 语句中链接多个上下文件管理器。

我们先来处理最后那一部分。

```
print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

这是一个复杂的 `with` 语句。让我改写它使之更有可读性。

```
with open('out.log', mode='w', encoding='utf-8') as a_file:
    with RedirectStdoutTo(a_file):
        print('B')
```

正如改动后的代码所展示的，实际上你使用了两个 `with` 语句，其中一个嵌套在另外一个的作用域(scope)里。“外层的” `with` 语句你应该已经熟悉了：它打开一个使用UTF-8编码的叫做 `out.log` 的文本文件用来写入，然后把返回的流对象赋给一个叫做 `a_file` 的变量。但是，在此处，它并不是唯一显得古怪的事情。

```
with RedirectStdoutTo(a_file):
```

`as` 子句(clause)到哪里去了？其实 `with` 语句并不一定需要 `as` 子句。就像你调用一个函数然后忽略其返回值一样，你也可以不把 `with` 语句的上下文环境赋给一个变量。在这种情况下，我们只关心 `RedirectStdoutTo` 上下文环境的边际效应(side effect)。``

`那么，这些边际效应都是些什么呢？我们来看一看 RedirectStdoutTo 类的内部结构。这是一个用户自定义的[上下文
和 `__exit__()` 就可以变成上下文管理器。`

```
`class RedirectStdoutTo:

        self.out_new = out_new

        self.out_old = sys.stdout
        sys.stdout = self.out_new

        sys.stdout = self.out_old
```

1. 在实例被创建后 `__init__()` 方法马上被调用。它使用一个参数，即在上下文环境的生命周期内你想用做标准输出的流对象。这个方法只是把该流对象保存在一个实例变量里(instance variable)以使其他方法在后边能够使用到它。
2. `__enter__()` 方法是一个特殊的类方法(special class method)；在进入一个上下文环境时Python会调用它（即，在 `with` 语句的开始处）。该方法把当前 `sys.stdout` 的值保存在 `self.out_old` 内，然后通过把 `self.out_new` 赋给 `sys.stdout` 来重定向标准输出。
3. `__exit__()` 是另外一个特殊类方法；当离开一个上下文环境时（即，在 `with` 语句的末尾）Python会调用它。这个方法通过把保存的 `self.out_old` 的值赋给 `sys.stdout` 来恢复标准输出到原来的状态。

放到一起：

1. 这条代码会输出到IDE的“交互式窗口(Interactive Window)”（或者终端，如果你从命令行运行这段脚本）。
2. 这条 `with` 语句使用逗号分隔的上下文环境列表。这个列表就像一系列相互嵌套的 `with` 块。先列出的是“外层”的块；后列出的是“内层”的块。第一个上下文环境打开一个文件；第二个重定向 `sys.stdout` 到由第一个上下环境创建的流对象。
3. 由于这个 `print()` 函数在 `with` 语句创建的上下文环境里执行，所以它不会输出到屏幕；它会写入到文件 `out.log` 。
4. `with` 语句块结束了。Python告诉每一个上下文管理器完成他们应该在离开上下文环境时应该做的事。这些上下文环境形成一个后进先出的栈。当离开一个上下文环境的时候，第二个上下文环境将 `sys.stdout` 的值恢复到它的原来状态，然后第一个上下文环境关闭

> 那个叫做 `out.log` 的文件。由于标准输出已经被恢复到原来的状态，再次调用 `print()` 函数会马上输出到屏幕上。

重定向标准错误的原理跟这个完全一样，将 `sys.stdout` 替换为 `sys.stderr` 即可。

进一步阅读

- 读写文件 Python.org 上的教程
- `io` 模块
- 流对象
- 上下文管理器类型
- `sys.stdout` and `sys.stderr`
- FUSE 来自维基百科

Chapter 12 XML

> " In the archonship of Aristaechmus, Draco enacted his ordinances. " — Aristotle

概述

这本书的大部分章节都是以样例代码为中心的。但是XML这章不是；它以数据为中心。最常见的XML应用为“聚合订阅(syndication feeds)”，它用来展示博客，论坛或者其他会经常更新的网站的最新内容。大多数的博客软件都会在新文章，新的讨论区，或者新博文发布的时候自动生成和更新feed。我们可以通过“订阅(subscribe)”feed来关注它们，还可以使用专门的“feed聚合工具(feed aggregator)”，比如Google Reader。

以下的XML数据是我们这一章中要用到的。它是一个feed — 更确切地说是一个Atom聚合feed

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  <link rel='self' type='application/atom+xml' href='http://diveintomark.org/feed/'/>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Dive into history, 2009 edition</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'/>
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    <updated>2009-03-27T21:56:07Z</updated>
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython'/>
    <category scheme='http://diveintomark.org' term='docbook'/>
    <category scheme='http://diveintomark.org' term='html'/>
  <summary type='html'>Putting an entire chapter on one page sounds
    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds&amp;hellip;
    On dialup.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Accessibility is a harsh mistress</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress
    <id>tag:diveintomark.org,2009-03-21:/archives/20090321200928</id>
    <updated>2009-03-22T01:05:37Z</updated>
    <published>2009-03-21T20:09:28Z</published>
    <category scheme='http://diveintomark.org' term='accessibility'/>
    <summary type='html'>The accessibility orthodoxy does not permit people to
      question the value of features that are rarely useful and rarely used.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
    </author>
    <title>A gentle introduction to video encoding, part 1: container formats</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats'/>
    <id>tag:diveintomark.org,2008-12-18:/archives/20081218155422</id>
    <updated>2009-01-11T19:39:22Z</updated>
    <published>2008-12-18T15:54:22Z</published>
    <category scheme='http://diveintomark.org' term='asf'/>
    <category scheme='http://diveintomark.org' term='avi'/>
    <category scheme='http://diveintomark.org' term='encoding'/>
    <category scheme='http://diveintomark.org' term='flv'/>
    <category scheme='http://diveintomark.org' term='GIVE'/>
    <category scheme='http://diveintomark.org' term='mp4'/>
    <category scheme='http://diveintomark.org' term='ogg'/>
    <category scheme='http://diveintomark.org' term='video'/>
    <summary type='html'>These notes will eventually become part of a
      tech talk on video encoding.</summary>
  </entry>
</feed>
```

5分钟XML速成

如果你已经了解XML，可以跳过这一部分。

XML是一种描述层次结构化数据的通用方法。XML文档包含由起始和结束标签*(tag)*分隔的一个或多个元素*(element)*。以下也是一个完整的(虽然空洞)XML文件：

1. 这是 `foo` 元素的起始标签。
2. 这是 `foo` 元素对应的结束标签。就如写作、数学或者代码中需要平衡括号一样，每一个起始标签必须有对应的结束标签来闭合（匹配）。

元素可以嵌套到任意层次。位于 `foo` 中的元素 `bar` 可以被称作其子元素。

```
<foo>
  <mark><bar></bar></mark>
</foo>
```

XML文档中的第一个元素叫做根元素*(root element)*。并且每份XML文档只能有一个根元素。以下不是一个XML文档，因为它存在两个“根元素”。

```
<foo></foo>
<bar></bar>
```

元素可以有其属性*(attribute)*，它们是一些名字-值(name-value)对。属性由空格分隔列举在元素的起始标签中。一个元素中属性名不能重复。属性值必须用引号包围起来。单引号、双引号都是可以。

```
</foo>
```

1. `foo` 元素有一个叫做 `lang` 的属性。 `lang` 的值为 `en`
2. `bar` 元素则有两个属性，分别为 `id` 和 `lang` 。其中 `lang` 属性的值为 `fr` 。它不会与 `foo` 的那个属性产生冲突。每个元素都其独立的属性集。

如果元素有多个属性，书写的顺序并不重要。元素的属性是一个无序的键-值对集，跟Python中的列表对象一样。另外，元素中属性的个数是没有限制的。

元素可以有其文本内容*(text content)*

```
<foo lang='en'>
  <bar lang='fr'><mark>PapayaWhip</mark></bar>
</foo>
```

如果某一元素既没有文本内容，也没有子元素，它也叫做空元素。

```
<foo></foo>
```

表达空元素有一种简洁的方法。通过在起始标签的尾部添加 `/` 字符，我们可以省略结束标签。上一个例子中的XML文档可以写成这样：

```
<foo<mark>/</mark>>
```

就像Python函数可以在不同的模块*(modules)*中声明一样，也可以在不同的名字空间*(namespace)*中声明XML元素。XML文档的名字空间通常看起来像URL。我们可以通过声明 `xmlns` 来定义默认名字空间。名字空间声明跟元素属性看起来很相似，但是它们的作用是不一样的。

```
</feed>
```

1. `feed` 元素处在名字空间 `http://www.w3.org/2005/Atom` 中。
2. `title` 元素也是。名字空间声明不仅会作用于当前声明它的元素，还会影响到该元素的所有子元素。

也可以通过 `xmlns:` prefix `声明来定义一个名字空间并取其名为_prefix_。然后该名字空间中的每个元素都必须显式地使用这个 prefix`)来声明。

```
</atom:feed>
```

1. `feed` 元素属于名字空间 `http://www.w3.org/2005/Atom` 。
2. `title` 元素也在那个名字空间。

对于XML解析器而言，以上两个XML文档是一样的。名字空间 + 元素名 = XML标识。前缀只是用来引用名字空间的，所以对于解析器来说，这些前缀名(`atom:`)其实无关紧要的。名字空间相同，元素名相同，属性（或者没有属性）相同，每个元素的文本内容相同，则XML文档相同。

最后，在根元素之前，字符编码信息可以出现在XML文档的第一行。（这里存在一个两难的局面(catch-22)，直观上来说，解析XML文档需要这些编码信息，而这些信息又存在于XML文档中，如果你对XML如何解决此问题有兴趣，请参阅XML规范中 F 章节）

```
<?xml version='1.0' <mark>encoding='utf-8'</mark>?>
```

现在我们已经知道足够多的XML知识，可以开始探险了！

Atom Feed的结构

想像一下网络上的博客，或者互联网上任何需要频繁更新的网站，比如CNN.com。该站点有一个标题(“CNN.com”)，一个子标题(“Breaking News, U.S., World, Weather, Entertainment & Video News”)，包含上次更新的日期(“updated 12:43 p.m. EDT, Sat May 16, 2009”)，还有在不同时期发布的文章的列表。每一篇文章也有自己的标题，第一次发布的日期（如果曾经修订过或者改正过某个输入错误，或许也有一个上次更新的日期），并且每篇文章有自己唯一的URL。

Atom聚合格式被设计成可以包含所有这些信息的标准格式。我的博客无论在设计，主题还是读者上都与CNN.com大不相同，但是它们的基本结构是相同的。CNN.com能做的事情，我的博客也能做…

每一个Atom订阅都共享着一个根元素：即在名字空间 `http://www.w3.org/2005/Atom` 中的元素 `feed` 。

1. `http://www.w3.org/2005/Atom` 表示名字空间Atom。
2. 每一个元素都可以包含 `xml:lang` 属性，它用来声明该元素及其子元素使用的语言。在当前样例中， `xml:lang` 在根元素中被声明了一次，也就意味着，整个feed都使用英文。

描述Atom feed自身的一些信息在根元素 `feed` 的子元素中被声明。

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
```

1. 该行表示这个feed的标题为 `dive into mark` 。
2. 这一行表示子标题为 `currently between addictions` 。
3. 每一个feed都要有一个全局唯一标识符(globally unique identifier)。想要知道如何创建它，请查阅RFC 4151。
4. 表示当前feed上次更新的时间为March 27, 2009, at 21:56 GMT。通常来说，它与最近一篇文章最后一次被修改的时间是一样的。
5. 事情开始变得有趣了… `link` 元素没有文本内容，但是它有三个属性： `rel` ， `type` 和 `href` 。 `rel` 元素的值能告诉我们链接的类型； `rel='alternate'` 表示这个链接指向当前feed的另外一个版本。 `type='text/html'` 表示链接的目标是一个HTML页面。然后目标地址在 `href` 属性中指出。

现在我们知道这个feed上一更新是在on March 27, 2009，它是为一个叫做“dive into mark”的站点准备的，并且站点的地址为 `http://diveintomark.org/` 。

> ☞在有一些XML文档中，元素的排列顺序是有意义的，但是Atom feed中不需要这样做。

feed级的元数据后边就是最近文章的列表了。单独的一篇文章就像这样：

```
<entry>

    <name>Mark</name>
    <uri>http://diveintomark.org/</uri>
  </author>

    href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'/>

  <published>2009-03-27T17:20:42Z</published>

  <category scheme='http://diveintomark.org' term='docbook'/>
  <category scheme='http://diveintomark.org' term='html'/>

    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds&amp;hellip;
    On dialup.</summary>
```

1. `author` 元素指示文章的作者：一个叫做Mark的伙计，并且我们可以在 `http://diveintomark.org/` 找到他的事迹。（这就像是feed元素里的备用链接，但是没有规定一定要这样。许多网络日志由多个作者完成，他们都有自己的个人主页。）* `title` 元素给出这篇文章的标题，即“Dive into history, 2009 edition”。
2. `如 feed 元素中的备用链接一样，` link `元素给出这篇文章的HTML` 版本地址。
3. 每个条目也像feed一样，需要一个唯一的标识。
4. 每个条目有两个日期与其相关：第一次发布日期(`published`)和上次修改日期(`updated`)。
5. 条目可以属于任意多个类别。这篇文章被归类到 `diveintopython` ， `docbook` ， 和 `html` 。
6. `summary` 元素中有这篇文章的概要性描述。（还有一个元素这里没有展示出来，即 `content` ，我们可以把整篇文章的内容都放在里边。）当前样例中， `summary` 元素含有一个Atom特有的 `type='html'` 属性，它用来告知这份概要为HTML格式，而非纯文本。这非常重要，因为概要内容中包含了HTML中特有的实体（ `—` 和 `…` ），它们不应该以纯文本直接显示，正确的形式应该为“—”和“…”。
7. 最后就是 `entry` 元素的结束标记了，它指示文章元数据的结尾。

解析XML

Python可以使用几种不同的方式解析XML文档。它包含了DOM和SAX解析器，但是我们焦点将放在另外一个叫做ElementTree的库上边。

```
<Element {http://www.w3.org/2005/Atom}feed at cd1eb0>
```

1. ElementTree属于Python标准库的一部分，它的位置为 `xml.etree.ElementTree` 。
2. `parse()` 函数是ElementTree库的主要入口，它使用文件名或者流对象作为参数。 `parse()` 函数会立即解析完整个文档。如果内存资源紧张，也可以增量式地解析XML文档
3. `parse()` 函数会返回一个能代表整篇文档的对象。这不是根元素。要获得根元素的引用

可以调用 `getroot()` 方法。
4. 如预期的那样，根元素即 `http://www.w3.org/2005/Atom` 名字空间中的 `feed` 。该字符串表示再次重申了非常重要的一点：XML元素由名字空间和标签名（也称作本地名*(local name)*）组成。这篇文档中的每个元素都在名字空间Atom中，所以根元素被表示为 `{http://www.w3.org/2005/Atom}feed` 。

> ☞ElementTree使用 `{` namespace `}` localname``来表达XML元素。我们将会在ElementTree的API中多次见到这种形式。

元素即列表

在ElementTree API中，元素的行为就像列表一样。列表中的项即该元素的子元素。

```
# continued from the previous example

'{http://www.w3.org/2005/Atom}feed'

8

...
<Element {http://www.w3.org/2005/Atom}title at e2b5d0>
<Element {http://www.w3.org/2005/Atom}subtitle at e2b4e0>
<Element {http://www.w3.org/2005/Atom}id at e2b6c0>
<Element {http://www.w3.org/2005/Atom}updated at e2b6f0>
<Element {http://www.w3.org/2005/Atom}link at e2b4b0>
<Element {http://www.w3.org/2005/Atom}entry at e2b720>
<Element {http://www.w3.org/2005/Atom}entry at e2b510>
<Element {http://www.w3.org/2005/Atom}entry at e2b750>
```

1. 紧接前一例子，根元素为 `{http://www.w3.org/2005/Atom}feed` 。
2. 根元素的“长度”即子元素的个数。
3. 我们可以像使用迭代器一样来遍历其子元素。
4. 从输出可以看到，根元素总共有8个子元素：所有feed级的元数据（ `title` ， `subtitle` ， `id` ， `updated` 和 `link` ），还有紧接着的三个 `entry` 元素。

也许你已经注意到了，但我还是想要指出来：该列表只包含直接子元素。每一个 `entry` 元素都有其子元素，但是并没有包括在这个列表中。这些子元素本可以包括在 `entry` 元素的列表中，但是确实不属于 `feed` 的子元素。但是，无论这些元素嵌套的层次有多深，总是有办法定位到它们的；在这章的后续部分我们会介绍两种方法。

属性即字典

XML不只是元素的集合；每一个元素还有其属性集。一旦获取了某个元素的引用，我们可以像操作Python的字典一样轻松获取到其属性。

```
# continuing from the previous example

{'{http://www.w3.org/XML/1998/namespace}lang': 'en'}

<Element {http://www.w3.org/2005/Atom}link at e181b0>

{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}

<Element {http://www.w3.org/2005/Atom}updated at e2b4e0>

{}
```

1. `attrib` 是一个代表元素属性的字典。这个地方原来的标记语言是这样描述的：`<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>`。前缀 `xml:` 指示一个内置的名字空间，每一个XML不需要声明就可以使用它。
2. 第五个子元素 — 以0为起始的列表中即 `[4]` — 为元素 `link`。
3. `link` 元素有三个属性：`href`，`type`，和 `rel`。
4. 第四个子元素 — `[3]` — 为 `updated`。
5. 元素 `updated` 没有子元素，所以 `.attrib` 是一个空的字典对象。

在XML文档中查找结点

到目前为止，我们已经“自顶向下“地从根元素开始，一直到其子元素，走完了整个文档。但是许多情况下我们需要找到XML中特定的元素。Etree也能完成这项工作。

```
>>> import xml.etree.ElementTree as etree
>>> tree = etree.parse('examples/feed.xml')
>>> root = tree.getroot()

[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> root.tag
'{http://www.w3.org/2005/Atom}feed'

[]

[]
```

1. `findfall()` 方法查找匹配特定格式的子元素。（关于查询的格式稍后会讲到。）
2. 每个元素 — 包括根元素及其子元素 — 都有 `findall()` 方法。它会找到所有匹配的子元素。但是为什么没有看到任何结果呢？也许不太明显，这个查询只会搜索其子元素。由于根元素 `feed` 中不存在任何叫做 `feed` 的子元素，所以查询的结果为一个空的列表。
3. 这个结果也许也在你的意料之外。在这篇文档中确实存在 `author` 元素；事实上总共有三个（每个 `entry` 元素中都有一个）。但是那些 `author` 元素不是根元素的直接子元素。我们可以在任意嵌套层次中查找 `author` 元素，但是查询的格式会有些不同。

```
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]

[]
```

1. 为了方便，对象 `tree` （调用 `etree.parse()` 的返回值）中的一些方法是根元素中这些方法的镜像。在这里，如果调用 `tree.getroot().findall()` ，则返回值是一样的。
2. 也许有些意外，这个查询请求也没有找到文档中的 `author` 元素。为什么没有呢？因为它只是 `tree.getroot().findall('{http://www.w3.org/2005/Atom}author')` 的一种简洁表示，即“查询所有是根元素的子元素的 `author` ”。因为这些 `author` 是 `entry` 元素的子元素，所以查询没有找到任何匹配的。

`find()` 方法用来返回第一个匹配到的元素。当我们认为只会有一个匹配，或者有多个匹配但我们只关心第一个的时候，这个方法是很有用的。

```
>>> len(entries)
3

>>> title_element.text
'Dive into history, 2009 edition'

>>> foo_element
>>> type(foo_element)
<class 'NoneType'>
```

1. 在前一样例中已经看到。这一句返回所有的 `atom:entry` 元素。
2. `find()` 方法使用ElementTree作为参数，返回第一个匹配到的元素。
3. 在 `entries[0]` 中没有叫做 `foo` 的元素，所以返回值为 `None` 。

> ☞可逮住你了，在这里 `find()` 方法非常容易被误解。在布尔上下文中，如果ElementTree元素对象不包含子元素，其值则会被认为是 `False` （即如果 `len(element)` 等于0）。这就意味着 `if element.find('...')` 并非在测试是否 `find()` 方法找到了匹配项；这条语句是在测试匹配到的元素是否包含子元素！想要测试 `find()` 方法是否返回了一个元素，则需使用 `if element.find('...') is not None` 。

也可以在所有派生*(descendant)*元素中搜索，即任意嵌套层次的子元素，孙子元素等…

```
>>> all_links
[<Element {http://www.w3.org/2005/Atom}link at e181b0>,
 <Element {http://www.w3.org/2005/Atom}link at e2b570>,
 <Element {http://www.w3.org/2005/Atom}link at e2b480>,
 <Element {http://www.w3.org/2005/Atom}link at e2b5a0>]

{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}

{'href': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[2].attrib
{'href': 'http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[3].attrib
{'href': 'http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats',
 'type': 'text/html',
 'rel': 'alternate'}
```

1. `//{http://www.w3.org/2005/Atom}link` 与前一样例很相似，除了开头的两条斜线。这两条斜线告诉 `findall()` 方法“不要只在直接子元素中查找；查找的范围可以是任意嵌套层次”。
2. 查询到的第一个结果是根元素的直接子元素。从它的属性中可以看出，它是一个指向该feed的HTML版本的备用链接。
3. 其他的三个结果分别是低一级的备用链接。每一个 `entry` 都有单独一个 `link` 子元素，由于在查询语句前的两条斜线的作用，我们也能定位到他们。

总的来说，ElementTree的 `findall()` 方法是其一个非常强大的特性，但是它的查询语言却让人有些出乎意料。官方描述它为“有限的XPath支持。”XPath是一种用于查询XML文档的W3C标准。对于基础地查询来说，ElementTree与XPath语法上足够相似，但是如果已经会XPath的话，它们之间的差异可能会使你感到不快。现在，我们来看一看另外一个第三方XML库，它扩展了ElementTree的API以提供对XPath的全面支持。

深入lxml

`lxml` 是一个开源的第三方库，以流行的libxml2 解析器为基础开发。提供了与ElementTree完全兼容的API，并且扩展它以提供了对XPath 1.0的全面支持，以及改进了一些其他精巧的细节。提供Windows的安装程序；Linux用户推荐使用特定发行版自带的工具比如 `yum` 或者 `apt-get` 从它们的程序库中安装预编译好了的二进制文件。要不然，你就得手工安装他们了。

```
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
```

1. 导入 `lxml` 以后，可以发现它与内置的ElementTree库提供相同的API。
2. `parse()` 函数：与ElementTree相同。
3. `getroot()` 方法：相同。
4. `findall()` 方法：完全相同。

对于大型的XML文档， `lxml` 明显比内置的ElementTree快了许多。如果现在只用到了ElementTree的API，并且想要使用其最快的实现(implementation)，我们可以尝试导入 `lxml` ，并且将内置的ElementTree作为备用。

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

但是 `lxml` 不只是一个更快速的ElementTree。它的 `findall()` 方法能够支持更加复杂的表达式。

```

>>> tree = lxml.etree.parse('examples/feed.xml')

[<Element {http://www.w3.org/2005/Atom}link at eeb8a0>,
 <Element {http://www.w3.org/2005/Atom}link at eeb990>,
 <Element {http://www.w3.org/2005/Atom}link at eeb960>,
 <Element {http://www.w3.org/2005/Atom}link at eeb9c0>]

[<Element {http://www.w3.org/2005/Atom}link at eeb930>]
>>> NS = '{http://www.w3.org/2005/Atom}'

[<Element {http://www.w3.org/2005/Atom}author at eeba80>,
 <Element {http://www.w3.org/2005/Atom}author at eebba0>]
```

1. 在这个样例中，我使用了 `import lxml.etree` （而非 `from lxml import etree` ），以强调这些特性只限于 `lxml` 。
2. 这一句在整个文档范围内搜索名字空间Atom中具有 `href` 属性的所有元素。在查询语句开头的 `//` 表示"搜索的范围为整个文档（不只是根元素的子元素）。" `{http://www.w3.org/2005/Atom}` 指示"搜索范围仅在名字空间Atom中。" `*` 表示"任意本地名(local name)的元素。" `[@href]` 表示"含有 `href` 属性。"
3. 该查询找出所有包含 `href` 属性并且其值为 `http://diveintomark.org/` 的Atom元素。
4. 在简单的字符串格式化后（要不然这条复合查询语句会变得特别长），它搜索名字空间Atom中包含 `uri` 元素作为子元素的 `author` 元素。该条语句只返回了第一个和第二个 `entry` 元素中的 `author` 元素。最后一个 `entry` 元素中的 `author` 只包含有 `name` 属性，没有 `uri` 。

仍然不够用？ `lxml` 也集成了对任意XPath 1.0表达式的支持。我们不会深入讲解XPath的语法；那可能需要一整本书！但是我会给你展示它是如何集成到 `lxml` 去的。

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed.xml')

...     namespaces=NSMAP)

[<Element {http://www.w3.org/2005/Atom}entry at e2b630>]
>>> entry = entries[0]

['Accessibility is a harsh mistress']
```

1. 要查询名字空间中的元素，首先需要定义一个名字空间前缀映射。它就是一个Python字典对象。
2. 这就是一个XPath查询请求。这个XPath表达式目的在于搜索 `category` 元素，并且该元素包含有值为 `accessibility` 的 `term` 属性。但是那并不是查询的结果。请看查询字符串的尾端；是否注意到了 `/..` 这一块？它的意思是，“然后返回已经找到的 `category` 元素的父元素。”所以这条XPath查询语句会找到所有包含 `<category term='accessibility'>` 作为子元素的条目。
3. `xpath()` 函数返回一个ElementTree对象列表。在这篇文档中，只有一个 `category` 元素，并且它的 `term` 属性值为 `accessibility` 。
4. XPath表达式并不总是会返回一个元素列表。技术上说，一个解析了的XML文档的DOM模型并不包含元素；它只包含结点*(node)*。依据它们的类型，结点可以是元素，属性，甚至是文本内容。XPath查询的结果是一个结点列表。当前查询返回一个文本结点列表： `title` 元素(`atom:title`)的文本内容(`text()`)，并且 `title` 元素必须是当前元素的子元素(`./`)。

生成XML

Python对XML的支持不只限于解析已存在的文档。我们也可以从头来创建XML文档。

```
>>> import xml.etree.ElementTree as etree

<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

1. 实例化 `Element` 类来创建一个新元素。可以将元素的名字（名字空间 + 本地名）作为其第一个参数。当前语句在Atom名字空间中创建一个 `feed` 元素。它将会成为我们文档的根元素。
2. 将属性名和值构成的字典对象传递给 `attrib` 参数，这样就可以给新创建的元素添加属性。请注意，属性名应该使用标准的ElementTree格式， `{` namespace `}` localname``。
3. 在任何时候，我们可以使用ElementTree的 `tostring()` 函数序列化任意元素（还有它的子元素）。

这种序列化结果有使你感到意外吗？技术上说，ElementTree使用的序列化方法是精确的，但却不是最理想的。在本章开头给出的XML样例文档中定义了一个默认名字空间*(default namespace)*(`xmlns='http://www.w3.org/2005/Atom'`)。对于每个元素都在同一个名字空间中的

文档 — 比如Atom feeds — 定义默认的名字空间非常有用，因为只需要声明一次名字空间，然后在声明每个元素的时候只需要使用其本地名即可（<feed>，<link>，<entry>）。除非想要定义另外一个名字空间中的元素，否则没有必要使用前缀。

对于XML解析器来说，它不会“注意”到使用默认名字空间和使用前缀名字空间的XML文档之间有什么不同。当前序列化结果的DOM为：

```
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

与下列序列化的DOM是一模一样的：

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

实际上唯一不同的只是第二个序列化短了几个字符长度。如果我们改动整个样例feed，使每一个起始和结束标签都有一个 `ns0:` 前缀，这将为每个起始标签增加 4 个字符 × 79 个标签 + 4 个名字空间声明本身用到的字符，总共320个字符。假设我们使用UTF-8编码，那将是320个额外的字节。（使用gzip压缩以后，大小可以降到21个字节，但是，21个字节也是字节。）也许对个人来说这算不了什么，但是对于像Atom feed这样的东西，只要稍有改变就有可能被下载上千次，每一个请求节约的几个字节就会迅速累加起来。

内置的ElementTree库没有提供细粒度地对序列化时名字空间内的元素的控制，但是 `lxml` 有这样的功能。

```
>>> import lxml.etree

<feed xmlns='http://www.w3.org/2005/Atom'/>

>>> print(lxml.etree.tounicode(new_feed))
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

1. 首先，定义一个用于名字空间映射的字典对象。其值为名字空间；字典中的键即为所需要的前缀。使用 `None` 作为前缀来定义默认的名字空间。
2. 现在我们可以在创建元素的时候，给 `lxml` 专有的 `nsmap` 参数传值，并且 `lxml` 会参照我们所定义的名字空间前缀。
3. 如所预期的那样，该序列化使用Atom作为默认的名字空间，并且在声明 `feed` 元素的时候没有使用名字空间前缀。
4. 啊噢… 我们忘了加上 `xml:lang` 属性。我们可以使用 `set()` 方法来随时给元素添加所需属性。该方法使用两个参数：标准ElementTree格式的属性名，然后，属性值。（该方法不是 `lxml` 特有的。在该样例中，只有 nsmap参数是 lxml 特有的，它用来控制序列化输出时名字空间的前缀。）

难道每个XML文档只能有一个元素吗？当然不了。我们可以创建子元素。

```
>>> print(lxml.etree.tounicode(new_feed))
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type='html'/></feed>

<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type='html'>dive into &amp

<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
<title type='html'>dive into&amp;hellip;</title>
</feed>
```

1. 给已有元素创建子元素，我们需要实例化 `SubElement` 类。它只要求两个参数，父元素（即该样例中的 `new_feed` ）和子元素的名字。由于该子元素会从父元素那儿继承名字空间的映射关系，所以这里不需要再声明名字空间前缀。
2. 我们也可以传递属性字典给它。字典的键即属性名；值为属性的值。
3. 如预期的那样，新创建的 `title` 元素在Atom名字空间中，并且它作为子元素插入到 `feed` 元素中。由于 `title` 元素没有文件内容，也没有其子元素，所以 `lxml` 将其序列化为一个空元素（使用 `/>` ）。
4. 设定元素的文本内容，只需要设定其 `.text` 属性。
5. 当前 `title` 元素序列化的时候就使用了其文本内容。任何包含了 `<` 或者 `&` 符号的内容在序列化的时候需要被转义。 `lxml` 会自动处理转义。
6. 我们也可以在序列化的时候应用“漂亮的输出(pretty printing)”，这会在每个结束标签的末尾，或者含有子元素但没有文本内容的标签的末尾添加换行符。用术语说就是， `lxml` 添加“无意义的空白(insignificant whitespace)”以使输出更具可读性。

> ☞你也许也想要看一看xmlwitch，它也是用来生成XML的另外一个第三方库。它大量地使用了 `with` 语句来使生成的XML代码更具可读性。

解析破损的XML

XML规范文档中指出，要求所有遵循XML规范的解析器使用“严厉的(draconian)错误处理”。即，当它们在XML文档中检测到任何编排良好性(wellformedness)错误的时候，应当立即停止解析。编排良好性错误包括不匹配的起始和结束标签，未定义的实体(entity)，非法的Unicode字符，还有一些只有内行才懂的规则(esoteric rules)。这与其他的常见格式，比如HTML，形成了鲜明的对比 — 即使忘记了封闭HTML标签，或者在属性值中忘了转义 `&` 字符，我们的浏览器也不会停止渲染一个Web页面。（通常大家认为HTML没有错误处理机制，这是一个常见的误解。HTML的错误处理实际上被很好的定义了，但是它比“遇见第一个错误即停止”这种机制要复杂得多。）

一些人（包括我自己）认为XML的设计者强制实行这种严格的错误处理本身是一个失误。请不要误解我；我当然能看到简化错误处理机制的优势。但是在现实中，“编排良好性”这种构想比乍听上去更加复杂，特别是对XML（比如Atom feeds）这种发布在网络上，通过HTTP传播的文档。早在1997年XML就标准化了这种严厉的错误处理，尽管XML已经非常成熟，研究一直表明，网络上相当一部分的Atom feeds仍然存在着编排完整性错误。

所以，从理论上和实际应用两种角度来看，我有理由“不惜任何代价”来解析XML文档，即，当遇到编排良好性错误时，不会中断解析操作。如果你认为你也需要这样做，`lxml` 可以助你一臂之力。

以下是一个破损的XML文档的片断。其中的编排良好性错误已经被高亮标出来了。

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into <mark>&hellip;</mark></title>
...
</feed>
```

因为实体 `…` 并没有在XML中被定义，所以这算作一个错误。（它在HTML中被定义。）如果我们尝试使用默认的设置来解析该破损的feed，`lxml` 会因为这个未定义的实体而停下来。

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed-broken.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lxml.etree.pyx", line 2693, in lxml.etree.parse (src/lxml/lxml.etree.c:52591)
  File "parser.pxi", line 1478, in lxml.etree._parseDocument (src/lxml/lxml.etree.c:75665
  File "parser.pxi", line 1507, in lxml.etree._parseDocumentFromURL (src/lxml/lxml.etree.
  File "parser.pxi", line 1407, in lxml.etree._parseDocFromFile (src/lxml/lxml.etree.c:75
  File "parser.pxi", line 965, in lxml.etree._BaseParser._parseDocFromFile (src/lxml/lxml
  File "parser.pxi", line 539, in lxml.etree._ParserContext._handleParseResultDoc (src/lx
  File "parser.pxi", line 625, in lxml.etree._handleParseResult (src/lxml/lxml.etree.c:68
  File "parser.pxi", line 565, in lxml.etree._raiseParseError (src/lxml/lxml.etree.c:6812
lxml.etree.XMLSyntaxError: Entity 'hellip' not defined, line 3, column 28
```

为了解析该破损的XML文档，忽略它的编排良好性错误，我们需要创建一个自定义的XML解析器。

```
examples/feed-broken.xml:3:28:FATAL:PARSER:ERR_UNDECLARED_ENTITY: Entity 'hellip' not def
>>> tree.findall('{http://www.w3.org/2005/Atom}title')
[<Element {http://www.w3.org/2005/Atom}title at ead510>]
>>> title = tree.findall('{http://www.w3.org/2005/Atom}title')[0]

'dive into '

<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into </title>
.
. [rest of serialization snipped for brevity]
.
```

1. 实例化 `lxml.etree.XMLParser` 类来创建一个自定义的解析器。它可以使用许多不同的命名参数。在此，我们感兴趣的为 `recover` 参数。当它的值被设为 `True`，XML解析器会尽力尝试从编排良好性错误中“恢复”。
2. 为使用自定的解析器来处理XML文档，将对象 `parser` 作为第二个参数传递给 `parse()` 函数。注意，`lxml` 没有因为那个未定义的 `…` 实体而抛出异常。

3. 解析器会记录它所遇到的所有编排良好性错误。（无论它是否被设置为需要从错误中恢复，这个记录总会存在。）
4. 由于不知道如果处理该未定义的 `…` 实体，解析器默认会将其省略掉。 `title` 元素的文本内容变成了 `'dive into '` 。
5. 从序列化的结果可以看出，实体 `…` 并没有被移到其他地方去；它就是被省略了。

在此，必须反复强调，这种“可恢复的”XML解析器没有互用性**(interoperability)**保证。另一个不同的解析器可能就会认为 `…` 来自HTML，然后将其替换为 `…` 。这样“更好”吗？也许吧。这样“更正确”吗？不，两种处理方法都不正确。正确的行为（根据XML规范）应该是终止解析操作。如果你已经决定不按规范来，你得自己负责。

进一步阅读

- 维基百科上的词条　XML
- ElementTree的XML API
- 元素和树状元素
- ElementTree中对XPath的支持
- ElementTree的迭代式解析(iterparse)功能
- `lxml`
- 使用 `lxml` 解析XML和HTML with
- 使用 `lxml` 解析XPath和XSLT
- xmlwitch

Chapter 13 序列化Python对象

> " Every Saturday since we've lived in this apartment, I have awakened at 6:15, poured myself a bowl of cereal, added a quarter-cup of 2% milk, sat on **this** end of **this** couch, turned on BBC America, and watched Doctor Who. " — Sheldon, The Big Bang Theory

深入

序列化的概念很简单。内存里面有一个数据结构，你希望将它保存下来，重用，或者发送给其他人。你会怎么做？嗯, 这取决于你想要怎么保存，怎么重用，发送给谁。很多游戏允许你在退出的时候保存进度，然后你再次启动的时候回到上次退出的地方。(实际上, 很多非游戏程序也会这么干。) 在这个情况下, 一个捕获了当前进度的数据结构需要在你退出的时候保存到磁盘上，接着在你重新启动的时候从磁盘上加载进来。这个数据只会被创建它的程序使用，不会发送到网络上，也不会被其它程序读取。因此，互操作的问题被限制在保证新版本的程序能够读取以前版本的程序创建的数据。

在这种情况下， `pickle` 模块是理想的。它是Python标准库的一部分, 所以它总是可用的。它很快; 它的大部分同Python解释器本身一样是用C写的。 它可以存储任意复杂的Python数据结构。

什么东西能用 `pickle` 模块存储?

- 所有Python支持的 原生类型 : 布尔, 整数, 浮点数, 复数, 字符串, `bytes` (字节串)对象, 字节数组, 以及 `None` .
- 由任何原生类型组成的列表，元组，字典和集合。
- 由任何原生类型组成的列表，元组，字典和集合组成的列表，元组，字典和集合(可以一直嵌套下去，直至Python支持的最大递归层数).
- 函数，类，和类的实例(带警告)。

如果这还不够用， `pickle` 模块也是可扩展的。如果你对可扩展性有兴趣，请查看本章最后的进一步阅读小节中的链接。

本章例子的快速笔记

本章会使用两个Python Shell来讲故事。本章的例子都是一个单独的故事的一部分。当我演示 `pickle` 和 `json` 模块时，你会被要求在两个Python Shell中来回切换。

为了让事情简单一点，打开Python Shell 并定义下面的变量:

```
>>> shell = 1
```

保持该窗口打开。 现在打开另一个Python Shell 并定义下面下面的变量:

```
>>> shell = 2
```

贯穿整个章节，在每个例子中我会使用 `shell` 变量来标识使用的是哪个Python Shell。

保存数据到 Pickle 文件

`pickle` 模块的工作对象是数据结构。让我们来创建一个：

```
1

>>> entry['title'] = 'Dive into history, 2009 edition'
>>> entry['article_link'] = 'http://diveintomark.org/archives/2009/03/27/dive-into-histor
>>> entry['comments_link'] = None
>>> entry['internal_id'] = b'\xDE\xD5\xB4\xF8'
>>> entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> entry['published'] = True
>>> import time

>>> entry['published_date']
time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm
```

1. 在Python Shell #1 里面。
2. 想法是建立一个Python字典来表示一些有用的东西，比如一个Atom 供稿的entry。但是为了炫耀一下 `pickle` 模块我也想保证里面包含了多种不同的数据类型。不需要太关心这些值。
3. `time` 模块包含一个表示时间点(精确到1毫秒)的数据结构(`time_struct`)以及操作时间结构的函数。 `strptime()` 函数接受一个格式化过的字符串并将其转化成一个 `time_struct` 。这个字符串使用的是默认格式，但你可以通过格式化代码来控制它。查看 `time` 模块来获得更多细节。

这是一个很帅的Python 字典。让我们把它保存到文件。

```
1
>>> import pickle

...
```

1. 仍然在Python Shell #1 中。
2. 使用 `open()` 函数来打开一个文件。设置文件模式为 `'wb'` 来以二进制写模式打开文件。把它放入 `with` 语句中来保证在你完成的时候文件自动被关闭。
3. `pickle` 模块中的 `dump()` 函数接受一个可序列化的Python 数据结构, 使用最新版本的pickle协议将其序列化为一个二进制的，Python特定的格式， 并且保存到一个打开的文件里。

最后一句话很重要。

- `pickle` 模块接受一个Python数据结构并将其保存的一个文件。
- 要做到这样，它使用一个被称为“pickle协议”的东西序列化该数据结构。
- pickle 协议是Python特定的，没有任何跨语言兼容的保证。你很可能不能使用Perl, PHP, Java, 或者其他语言来对你刚刚创建的 `entry.pickle` 文件做任何有用的事情。
- 并非所有的Python数据结构都可以通过 `pickle` 模块序列化。随着新的数据类型被加入到Python语言中，pickle协议已经被修改过很多次了，但是它还是有一些限制。
- 由于这些变化，不同版本的Python的兼容性也没有保证。新的版本的Python支持旧的序列化格式，但是旧版本的Python不支持新的格式(因为它们不支持新的数据类型)。
- 除非你指定， `pickle` 模块中的函数将使用最新版本的pickle协议。这保证了你对可以被序列化的数据类型有最大的灵活度，但这也意味着生成的文件不能被不支持新版pickle协议的旧版本的Python读取。
- 最新版本的pickle协议是二进制格式的。请确认使用二进制模式来打开你的pickle文件,否则当你写入的时候数据会被损坏。

从Pickle文件读取数据

现在切换到你的第二个Python Shell — 即不是你创建 `entry` 字典的那个。

```
2

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import pickle

...

{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link':
 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_mi
 'published': True}
```

1. 这是Python Shell #2.
2. 这里没有 `entry` 变量被定义过。你在Python Shell #1 中定义了 `entry` 变量, 但是那是另一个拥有自己状态的完全不同的环境。
3. 打开你在Python Shell #1中创建的 `entry.pickle` 文件。 `pickle` 模块使用二进制数据格式，所以你总是应该使用二进制模式打开pickle文件。
4. `pickle.load()` 函数接受一个流对象, 从流中读取序列化后的数据，创建一个新的Python对象，在新的Python对象中重建被序列化的数据，然后返回新建的Python对象。
5. 现在 `entry` 变量是一个键和值看起来都很熟悉的字典。

`pickle.dump() / pickle.load()` 循环的结果是一个和原始数据结构等同的新的数据结构。

```
1

...

True

False

('diveintopython', 'docbook', 'html')
>>> entry2['internal_id']
b'\xDE\xD5\xB4\xF8'
```

1. 切换回Python Shell #1。
2. 打开 `entry.pickle` 文件。
3. 将序列化后的数据装载到一个新的变量, `entry2` 。
4. Python 确认两个字典, `entry` 和 `entry2` 是相等的。在这个shell里, 你从零开始构造了 `entry` , 从一个空字典开始然后手工给各个键赋值。你序列化了这个字典并将其保存在 `entry.pickle` 文件中。现在你从文件中读取序列化后的数据并创建了原始数据结构的一个完美复制品。
5. 相等和相同是不一样的。我说的是你创建了原始数据结构的一个完美复制品, 这没错。但它仅仅是一个复制品。
6. 我要指出 `'tags'` 键对应的值是一个元组, 而 `'internal_id'` 键对应的值是一个 `bytes` 对象。原因在这章的后面就会清楚了。

不使用文件来进行序列化

前一节中的例子展示了如果将一个Python对象序列化到磁盘文件。但如果你不想或不需要文件呢？你也可以序列化到一个内存中的 `bytes` 对象。

```
>>> shell
1

<class 'bytes'>

True
```

1. `pickle.dumps()` 函数(注意函数名最后的 `'s'`)执行和 `pickle.dump()` 函数相同的序列化。取代接受流对象并将序列化后的数据保存到磁盘文件, 这个函数简单的返回序列化的数据。
2. 由于pickle协议使用一个二进制数据格式, 所以 `pickle.dumps()` 函数返回 `bytes` 对象。
3. `pickle.loads()` 函数(再一次, 注意函数名最后的 `'s'`) 执行和 `pickle.load()` 函数一样的反序列化。取代接受一个流对象并去文件读取序列化后的数据, 它接受包含序列化后的数据的 `bytes` 对象, 比如 `pickle.dumps()` 函数返回的对象。
4. 最终结果是一样的: 原始字典的完美复制。

字节串和字符串又一次抬起了它们丑陋的头。

pickle协议已经存在好多年了，它随着Python本身的成熟也不断成熟。现在存在四个不同版本的pickle协议。

- Python 1.x 有两个pickle协议，一个基于文本的格式(“版本 0”) 以及一个二进制格式(“版本 1”).
- Python 2.3 引入了一个新的pickle协议(“版本 2”) 来处理Python 类对象的新功能。它是一个二进制格式。
- Python 3.0 引入了另一个pickle 协议 (“版本 3”)， 显式的支持 `bytes` 对象和字节数组。它是一个二进制格式。

你看, 字节串和字符串的区别又一次抬起了它们丑陋的头。 (如果你觉得惊奇，你肯定开小差了。) 在实践中这意味着, 尽管Python 3 可以读取版本 2 的pickle 协议生成的数据, Python 2 不能读取版本 3的协议生成的数据.

调试Pickle 文件

pickle 协议是长什么样的呢？让我们离开Python Shell一会会，来看一下我们创建的 `entry.pickle` 文件。

```
you@localhost:~/diveintopython3/examples$ ls -l entry.pickle
-rw-r--r-- 1 you  you  358 Aug  3 13:34 entry.pickle
you@localhost:~/diveintopython3/examples$ cat entry.pickle
comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq?qX publishedq?
XlinkXJhttp://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition
q   Xpublished_dateq
ctime
struct_time
?qRqXtitleqXDive into history, 2009 editionqu.
```

这不是很有用。你可以看见字符串，但是其他数据类型显示为不可打印的(或者至少是不可读的)字符。域之间没有明显的分隔符(比如跳格符或空格)。你肯定不希望来调试这样一个格式。

```
>>> shell
1
>>> import pickletools
>>> with open('entry.pickle', 'rb') as f:
...     pickletools.dis(f)
 0: \x80 PROTO      3
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: X        BINUNICODE 'published_date'
   25: q        BINPUT     1
   27: c        GLOBAL     'time struct_time'
   45: q        BINPUT     2
   47: (        MARK
   48: M            BININT2    2009
   51: K            BININT1    3
   53: K            BININT1    27
   55: K            BININT1    22
   57: K            BININT1    20
   59: K            BININT1    42
   61: K            BININT1    4
   63: K            BININT1    86
   65: J            BININT     -1
   70: t            TUPLE      (MARK at 47)
   71: q        BINPUT     3
   73: }        EMPTY_DICT
   74: q        BINPUT     4
   76: \x86     TUPLE2
   77: q        BINPUT     5
   79: R        REDUCE
   80: q        BINPUT     6
   82: X        BINUNICODE 'comments_link'
  100: q        BINPUT     7
  102: N        NONE
  103: X        BINUNICODE 'internal_id'
  119: q        BINPUT     8
  121: C        SHORT_BINBYTES '脷脵麓酶'
  127: q        BINPUT     9
  129: X        BINUNICODE 'tags'
  138: q        BINPUT     10
  140: X        BINUNICODE 'diveintopython'
  159: q        BINPUT     11
  161: X        BINUNICODE 'docbook'
  173: q        BINPUT     12
  175: X        BINUNICODE 'html'
  184: q        BINPUT     13
  186: \x87     TUPLE3
  187: q        BINPUT     14
  189: X        BINUNICODE 'title'
  199: q        BINPUT     15
  201: X        BINUNICODE 'Dive into history, 2009 edition'
  237: q        BINPUT     16
  239: X        BINUNICODE 'article_link'
  256: q        BINPUT     17
  258: X        BINUNICODE 'http://diveintomark.org/archives/2009/03/27/dive-into-history
  337: q        BINPUT     18
  339: X        BINUNICODE 'published'
  353: q        BINPUT     19
  355: \x88     NEWTRUE
  356: u        SETITEMS   (MARK at 5)
  357: .    STOP
<mark>highest protocol among opcodes = 3</mark>
```

这个反汇编中最有趣的信息是最后一行, 因为它包含了文件保存时使用的pickle协议的版本号。在pickle协议里面没有明确的版本标志。为了确定保存pickle文件时使用的协议版本，你需要查看序列化后的数据的标记(“opcodes”)并且使用硬编码的哪个版本的协议引入了哪些标

记的知识(来确定版本号)。 `pickle.dis()` 函数正是这么干的，并且它在反汇编的输出的最后一行打印出结果。下面是一个不打印，仅仅返回版本号的函数:

[下载 `pickleversion.py`]

```
import pickletools

def protocol_version(file_object):
    maxproto = -1
    for opcode, arg, pos in pickletools.genops(file_object):
        maxproto = max(maxproto, opcode.proto)
    return maxproto
```

实际使用它:

```
>>> import pickleversion
>>> with open('entry.pickle', 'rb') as f:
...     v = pickleversion.protocol_version(f)
>>> v
3
```

序列化Python对象以供其它语言读取

`pickle` 模块使用的数据格式是Python特定的。它没有做任何兼容其它编程语言的努力。如果跨语言兼容是你的需求之一，你得去寻找其它的序列化格式。一个这样的格式是JSON。"JSON" 代表 "JavaScript Object Notation," 但是不要让名字糊弄你。 — JSON 是被设计为跨语言使用的。

Python 3 在标准库中包含了一个 `json` 模块。同 `pickle` 模块类似, `json` 模块包含一些函数，可以序列化数据结构，保存序列化后的数据至磁盘，从磁盘上读取序列化后的数据，将数据反序列化成新的Pythone对象。但两者也有一些很重要的区别。 首先, JSON数据格式是基于文本的, 不是二进制的。RFC 4627 定义了JSON格式以及怎样将各种类型的数据编码成文本。比如，一个布尔值要么存储为5个字符的字符串 `'false'` ，要么存储为4个字符的字符串 `'true'` 。 所有的JSON值都是大小写敏感的。

第二，由于是文本格式, 存在空白(whitespaces)的问题。 JSON 允许在值之间有任意数目的空白(空格, 跳格, 回车，换行)。空白是"无关紧要的"，这意味着JSON编码器可以按它们的喜好添加任意多或任意少的空白, 而JSON解码器被要求忽略值之间的任意空白。这允许你"美观的打印（pretty-print）" 你的 JSON 数据, 通过不同的缩进层次嵌套值，这样你就可以在标准浏览器或文本编辑器中阅读它。Python 的 `json` 模块有在编码时执行美观打印（pretty-printing）的选项。

第三, 字符编码的问题是长期存在的。JSON 用纯文本编码数据, 但是你知道, "不存在纯文本这种东西。" JSON必须以Unicode 编码(UTF-32, UTF-16, 或者默认的, UTF-8)方式存储, RFC 4627的第3节 定义了如何区分使用的是哪种编码。

将数据保存至 JSON 文件

JSON 看起来非常像你在Javascript中手工定义的数据结构。这不是意外; 实际上你可以使用JavaScript 的 `eval()` 函数来"解码" JSON序列化过的数据。(通常的对非信任输入的警告 也适用, 但关键点是JSON 是 合法的JavaScript。) 因此, 你可能已经熟悉JSON了。

```
>>> shell
1

>>> basic_entry['id'] = 256
>>> basic_entry['title'] = 'Dive into history, 2009 edition'
>>> basic_entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> basic_entry['published'] = True
>>> basic_entry['comments_link'] = None
>>> import json
```

1. 我们将创建一个新的数据结构，而不是重用现存的 `entry` 数据结构。在这章的后面, 我们将会看见当我们试图用JSON编码更复杂的数据结构的时候会发生什么。
2. JSON 是一个基于文本的格式， 这意味你可以以文本模式打开文件，并给定一个字符编码。用UTF-8总是没错的。
3. 同 `pickle` 模块一样，`json` 模块定义了 `dump()` 函数， 它接受一个Python 数据结构和一个可写的流对象。 `dump()` 函数将Python数据结构序列化并写入到流对象中。在 `with` 语句内工作保证当我们完成的时候正确的关闭文件。

那么生成的JSON序列化数据是什么样的呢？

```
you@localhost:~/diveintopython3/examples$ cat basic.json
{"published": true, "tags": ["diveintopython", "docbook", "html"], "comments_link": null,
"id": 256, "title": "Dive into history, 2009 edition"}
```

这肯定定比pickle 文件更可读。然而 JSON 的值之间可以包含任意数目的空把, 并且 `json` 模块提供了一个方便的途径来利用这一点生成更可读的JSON文件。

```
>>> shell
1
>>> with open('basic-pretty.json', mode='w', encoding='utf-8') as f:
```

1. 如果你给 `json.dump()` 函数传入 `indent` 参数, 它以文件变大为代价使生成的JSON文件更可读。 `indent` 参数是一个整数。0 意味着"每个值单独一行。" 大于0的数字意味着"每个值单独一行并且使用这个数目的空格来缩进嵌套的数据结构。"

这是结果:

```
you@localhost:~/diveintopython3/examples$ cat basic-pretty.json
{
  "published": true,
  "tags": [
    "diveintopython",
    "docbook",
    "html"
  ],
  "comments_link": null,
  "id": 256,
  "title": "Dive into history, 2009 edition"
}
```

将Python数据类型映射到JSON

由于JSON 不是Python特定的，对应到Python的数据类型的时候有很多不匹配。有一些仅仅是名字不同，但是有两个Python数据类型完全缺少。看看你能能把它们指出来:

笔记	JSON	Python 3
	object	dictionary
	array	list
	string	string
	integer	integer
	real number	float
*	`true`	`True`
*	`false`	`False`
*	`null`	`[None](native-datatypes.html#none)`
*	所有的 JSON 值都是大小写敏感的。	

注意到什么被遗漏了吗？元组和 & 字节串（bytes）! JSON 有数组类型, `json` 模块将其映射到Python的列表, 但是它没有一个单独的类型对应 “冻结数组(frozen arrays)” (元组)。而且尽管 JSON 非常好的支持字符串，但是它没有对 `bytes` 对象或字节数组的支持。

序列化JSON不支持的数据类型

即使JSON没有内建的字节流支持, 并不意味着你不能序列化 `bytes` 对象。 `json` 模块提供了编解码未知数据类型的扩展接口。(“未知”的意思是≴JSON没有定义”。很显然 `json` 模块认识字节数组, 但是它被JSON规范的限制束缚住了。) 如果你希望编码字节串或者其它JSON没有原生支持的数据类型，你需要给这些类型提供定制的编码和解码器。

```
>>> shell
1

{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edit
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_mi
 'published': True}
>>> import json

...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "C:\Python31\lib\json\encoder.py", line 170, in default
    raise TypeError(repr(o) + " is not JSON serializable")
<mark>TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable</mark>
```

1. 好的, 是时间再看看 `entry` 数据结构了。它包含了所有的东西: 布尔值, `None` 值，字符串，字符串元组, `bytes` 对象, 以及 `time` 结构体。
2. 我知道我已经说过了，但是这值得再重复一次：JSON 是一个基于文本的格式。总是应使用UTF-8字符编码以文本模式打开JSON文件。
3. 嗯，这可不好。发生什么了？

情况是这样的: `json.dump()` 函数试图序列化 `bytes` 对象 `b'\xDE\xD5\xB4\xF8'` ，但是它失败了，原因是JSON 不支持 `bytes` 对象。然而, 如果保存字节串对你来说很重要，你可以定义自己的“迷你序列化格式。”

1. 为了给一个JSON没有原生支持的数据类型定义你自己的“迷你序列化格式”, 只要定义一个接受一个Python对象为参数的函数。这个对象将会是 `json.dump()` 函数无法自己序列化的实际对象 — 这个例子里是 `bytes` 对象 `b'\xDE\xD5\xB4\xF8'` 。
2. 你的自定义序列化函数应该检查 `json.dump()` 函数传给它的对象的类型。当你的函数只序列化一个类型的时候这不是必须的，但是它使你的函数的覆盖的内容清楚明白，并且在你需要序列化更多类型的时候更容易扩展。
3. 在这个例子里面, 我将 `bytes` 对象转换成字典。 `__class__` 键持有原始的数据类型(以字符串的形式, `'bytes'`), 而 `__value__` 键持有实际的数据。当然它不能是 `bytes` 对象; 大体的想法是将其转换成某些可以被JSON序列化的东西! `bytes` 对象就是一个范围在0–255的整数的序列。 我们可以使用 `list()` 函数将 `bytes` 对象转换成整数列表。所以 `b'\xDE\xD5\xB4\xF8'` 变成 `[222, 213, 180, 248]` . (算一下! 这是对的! 16进制的字节 `\xDE` 是十进制的 222, `\xD5` 是 213, 以此类推。)
4. 这一行很重要。你序列化的数据结构可能包含JSON内建的可序列化类型和你的定制序列化器支持的类型之外的东西。在这种情况下，你的定制序列化器抛出一个 `TypeError` ，

那样 `json.dump()` 函数就可以知道你的定制序列化函数不认识该类型。

就这么多；你不需要其它的东西。特别是, 这个定制序列化函数返回*Python*字典， 不是字符串。你不是自己做所有序列化到JSON的工作; 你仅仅在做转换成被支持的类型那部分工作。 `json.dump()` 函数做剩下的事情。

```
>>> shell
1

...
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
    json.dump(entry, f, default=customserializer.to_json)
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "/Users/pilgrim/diveintopython3/examples/customserializer.py", line 12, in to_json

TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm
```

1. `customserializer` 模块是你在前一个例子中定义 `to_json()` 函数的地方。
2. 文本模式, UTF-8 编码, yadda yadda。(你很可能会忘记这一点! 我就忘记过好几次! 事情一切正常直到它失败的时刻, 而它的失败很令人瞩目。)
3. 这是重点: 为了将定制转换函数钩子嵌入 `json.dump()` 函数, 只要将你的函数以 `default` 参数传入 `json.dump()` 函数。(万岁, Python里一切皆对象!)
4. 好吧, 实际上还是不能工作。但是看一下异常。 `json.dump()` 函数不再抱怨无法序列化 `bytes` 对象了。现在它在抱怨另一个完全不同的对象: `time.struct_time` 对象。

尽管得到另一个不同的异常看起来不是什么进步, 但它确实是个进步! 再调整一下就可以解决这个问题。

```
 import time

def to_json(python_object):

    if isinstance(python_object, bytes):
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' is not JSON serializable')
```

1. 在现存的 `customserializer.to_json()` 函数里面, 我们加入了Python 对象 (`json.dump()` 处理不了的那些) 是不是 `time.struct_time` 的判断。
2. 如果是的， 我们做一些同处理 `bytes` 对象时类似的事情来转换: 将 `time.struct_time` 结构转化成一个只包含JSON可序列化值的字典。在这个例子里, 最简单的将日期时间转换成JSON可序列化值的方法是使用 `time.asctime()` 函数将其转换成字符串。 `time.asctime()` 函数将难看的 `time.struct_time` 转换成字符串

`'Fri Mar 27 22:20:42 2009'` 。

有了两个定制的转换，整个 `entry` 数据结构序列化到JSON应该没有进一步的问题了。

```
>>> shell
1
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...
```

```
you@localhost:~/diveintopython3/examples$ ls -l example.json
-rw-r--r-- 1 you  you  391 Aug  3 13:34 entry.json
you@localhost:~/diveintopython3/examples$ cat example.json
{"published_date": {"__class__": "time.asctime", "__value__": "Fri Mar 27 22:20:42 2009"}
"comments_link": null, "internal_id": {"__class__": "bytes", "__value__": [222, 213, 180,
"tags": ["diveintopython", "docbook", "html"], "title": "Dive into history, 2009 edition"
"article_link": "http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-editi
"published": true}
```

从JSON文件加载数据

类似 `pickle` 模块， `json` 模块有一个 `load()` 函数接受一个流对象，从中读取 JSON编码过的数据，并且创建该JSON数据结构的Python对象的镜像。

```
>>> shell
2

>>> entry
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import json
>>> with open('entry.json', 'r', encoding='utf-8') as f:

...

{'comments_link': None,
 'internal_id': {'__class__': 'bytes', '__value__': [222, 213, 180, 248]},
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edit
 'published_date': {'__class__': 'time.asctime', '__value__': 'Fri Mar 27 22:20:42 2009'}
 'published': True}
```

1. 为了演示目的，切换到Python Shell #2 并且删除在这一章前面使用 `pickle` 模块创建的 `entry` 数据结构。
2. 最简单的情况下， `json.load()` 函数同 `pickle.load()` 函数的结果一模一样。你传入一个流对象，它返回一个新的Python对象。
3. 有好消息也有坏消息。好消息先来: `json.load()` 函数成功的读取了你在Python Shell #1 中创建的 `entry.json` 文件并且生成了一个包含那些数据的新的Python对象。接着是坏消息: 它没有重建原始的 `entry` 数据结构。 `'internal_id'` 和 `'published_date'` 这两个值

被重建为字典 — 具体来说, 你在 `to_json()` 转换函数中使用JSON兼容的值创建的字典。

`json.load()` 并不知道你可能传给 `json.dump()` 的任何转换函数的任何信息。你需要的是 `to_json()` 函数的逆函数 — 一个接受定制转换出的JSON 对象并将其转换回原始的Python数据类型。

```
# add this to customserializer.py

        if json_object['__class__'] == 'time.asctime':

        if json_object['__class__'] == 'bytes':

    return json_object
```

1. 这函数也同样接受一个参数返回一个值。但是参数不是字符串，而是一个Python对象 — 反序列化一个JSON编码的字符串为Python的结果。
2. 你只需要检查这个对象是否包含 `to_json()` 函数创建的 `'__class__'` 键。如果是的， `'__class__'` 键对应的值将告诉你如何将值解码成原来的Python数据类型。
3. 为了解码由 `time.asctime()` 函数返回的字符串，你要使用 `time.strptime()` 函数。这个函数接受一个格式化过的时间字符串(格式可以自定义，但默认值同 `time.asctime()` 函数的默认值相同) 并且返回 `time.struct_time` .
4. 为了将整数列表转换回 `bytes` 对象, 你可以使用 `bytes()` 函数。

就是这样; `to_json()` 函数处理了两种数据类型，现在这两个数据类型也在 `from_json()` 函数里面处理了。下面是结果:

```
>>> shell
2
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:

...

{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edit
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_mi
 'published': True}
```

1. 为了将 `from_json()` 函数嵌入到反序列化过程中，把它作为 `object_hook` 参数传入到 `json.load()` 函数中。接受函数作为参数的函数; 真方便!
2. `entry` 数据结构现在有一个值为 `bytes` 对象的 `'internal_id'` 键。它也包含一个 `'published_date'` 键，其值为 `time.struct_time` 对象。

然而，还有最后一个缺陷。

```
>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry2 = json.load(f, object_hook=customserializer.from_json)
...

False

('diveintopython', 'docbook', 'html')

['diveintopython', 'docbook', 'html']
```

1. 即使在序列化过程中加入了 `to_json()` 钩子函数, 也在反序列化过程中加入 `from_json()` 钩子函数, 我们仍然没有重新创建原始数据结构的完美复制品。为什么没有？
2. 在原始的 `entry` 数据结构中, `'tags'` 键的值为一个三个字符串组成的元组。
3. 但是重现创建的 `entry2` 数据结构中, `'tags'` 键的值是一个三个字符串组成的列表。JSON 并不区分元组和列表；它只有一个类似列表的数据类型，数组，并且 `json` 模块在序列化过程中会安静的将元组和列表两个都转换成JSON 数组。大多数情况下，你可以忽略元组和列表的区别，但是在使用 `json` 模块时应记得有这么一回使。

进一步阅读

> ☞很多关于 `pickle` 模块的文章提到了 `cPickle` 。在Python 2中, `pickle` 模块有两个实现, 一个由纯Python写的而另一个用C写的(但仍然可以在Python中调用)。在Python 3中, 这两个模块已经合并, 所以你总是简单的 `import pickle` 就可以。你可能会发现这些文章很有用，但是你应该忽略已过时的关于的 `cPickle` 的信息.

使用 `pickle` 模块打包:

- `pickle` module
- `pickle` and `cPickle` — Python object serialization
- Using `pickle`
- Python persistence management

使用JSON 和 `json` 模块:

- `json` — JavaScript Object Notation Serializer
- JSON encoding and ecoding with custom objects in Python

扩展打包:

- Pickling class instances
- Persistence of external objects
- Handling stateful objects

Chapter 14 HTTP Web 服务

> " A ruffled mind makes a restless pillow. " — Charlotte Brontë

深入

简单地讲，HTTP web 服务是指以编程的方式直接使用 HTTP 操作从远程服务器发送和接收数据。如果你要从服务器获取数据，使用HTTP `GET`；如果你要向服务器发送新数据，使用HTTP `POST`. 一些更高级的HTTP Web 服务 API也允许使用HTTP `PUT` 和 HTTP `DELETE` 来创建、修改和删除数据。 换句话说，HTTP 协议中的“verbs (动作)” (`GET`, `POST`, `PUT` 和 `DELETE`) 可以直接对应到应用层的操作：获取，创建，修改，删除数据。

这个方法主要的优点是简单, 它的简单证明是受欢迎的。数据 — 通常是XML或JSON — 可以事先创建好并静态的存储下来， 或者由服务器端脚本动态生成, 并且所有主要的编程语言(当然包括 Python)都包含HTTP 库用于下载数据。调试也很方便; 由于HTTP web 服务中每一个资源都有一个唯一的地址(以URL的形式存在), 你可以在浏览器中加载它并且立即看到原始的数据.

HTTP web 服务示例:

- Google Data API 允许你同很多类型的Google 服务交互, 包括 Blogger 和 YouTube。
- Flickr Services 允许你向Flickr下载和上传图片。
- Twitter API 允许你在Twitter发布状态更新。
- …以及更多

Python 3 带有两个库用于和HTTP web 服务交互:

- `http.client` 是实现了RFC 2616, HTTP 协议的底层库.
- `urllib.request` 建立在 `http.client` 之上一个抽象层。 它为访问HTTP 和 FTP 服务器提供了一个标准的API，可以自动跟随HTTP 重定向， 并且处理了一些常见形式的HTTP 认证。

那么，你应该用哪个呢？两个都不用。取而代之, 你应该使用 `httplib2` ,一个第三方的开源库, 它比 `http.client` 更完整的实现了HTTP协议，同时比 `urllib.request` 提供了更好的抽象。

要理解为什么 `httplib2` 是正确的选择，你必须先了解HTTP。

HTTP的特性

有五个重要的特性所有的HTTP客户端都应该支持。

缓存

关于web服务最需要了解的一点是网络访问是极端昂贵的。我并不是指“美元”和“美分”的昂贵(虽然带宽确实不是免费的)。我的意思是需要一个非常长的时间来打开一个连接，发送请求，并从远程服务器响应。 即使在最快的宽带连接上，延迟（从发送一个请求到开始在响应中获得数据所花费的时间）仍然高于您的预期。路由器的行为不端，被丢弃的数据包，中间代理服务器被攻击 — 在公共互联网上没有沉闷的时刻(never a dull moment)，并且你对此无能为力。

`Cache-Control: max-age` 的意思是“一个星期以内都不要来烦我。”

HTTP在设计时就考虑到了缓存。有这样一类的设备(叫做 “缓存代理服务器”)，它们的唯一的任务是就是呆在你和世界的其他部分之间来最小化网络请求。你的公司或ISP 几乎肯定维护着这样的缓存代理服务器, 只不过你没有意识到而已。 它们的能够起到作用是因为缓存是内建在HTTP协议中的。

这里有一个缓存如何工作的具体例子。 你通过浏览器访问 `diveintomark.org` 。该网页包含一个背景图片， `wearehugh.com/m.jpg` 。当你的浏览器下载那张图片时,服务器的返回包含了下面的HTTP 头:

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
<mark>Cache-Control: max-age=31536000, public</mark>
<mark>Expires: Mon, 31 May 2010 17:14:04 GMT</mark>
Connection: close
Content-Type: image/jpeg
```

`Cache-Control` 和 `Expires` 头告诉浏览器(以及任何处于你和服务器之间的缓存代理服务器)这张图片可以缓存长达一年。 一年! 如果在明年，你访问另外一个也包含这张图片的页面，你的浏览器会从缓存中加载这样图片而不会产生任何网络活动.

等一下，情况实际上更好。比方说，你的浏览器由于某些原因将图片从本地缓存中移除了。可能是因为没有磁盘空间了或者是你清空了缓存，不管是什么理由。然而HTTP 头告诉说这个数据可以被公共缓存代理服务器缓存(`Cache-Control` 头中 `public` 关键字说明这一点)。缓存代理服务器有非常庞大的存储空间，很可能比你本地浏览器所分配的大的多。

如果你的公司或者ISP维护着这样一个缓存代理服务器，它很可能仍然有这张图片的缓存。 当你再次访问 `diveintomark.org` 时, 你的浏览器会在本地缓存中查找这张图片, 它没有找到, 所以它发出一个网络请求试图从远程服务器下载这张图片。但是由于缓存代理服务器仍然有这张图片的一个副本，它将截取这个请求并从它的缓存中返回这张图片。 这意味这你的请求不会到达远程服务器; 实际上, 它根本没有离开你公司的网络。这意味着更快的下载(网络跃点变少了) 和节省你公司的花费(从外部下载的数据变少了)。

只有当每一个角色都做按协议来做时，HTTP缓存才能发挥作用。一方面，服务器需要在响应中发送正确的头。另一方面，客户端需要在第二次请求同样的数据前理解并尊重这些响应头。 代理服务器不是灵丹妙药，它们只会在客户端和服务器允许的情况下尽可能的聪明。

Python的HTTP库不支持缓存，而 `httplib2` 支持。

最后修改时间的检查

有一些数据从不改变，而另外一些则总是在变化。介于两者之间，在很多情况下数据还没变化但是将来可能会变化。 CNN.com 的供稿每隔几分钟就会更新，但我的博客的供稿可能几天或者几星期才会更新一次。在后面一种情况的时候，我不希望告诉客户端缓存我的供稿几星期，因为当我真的发表了点东西的时候，人们可能会几个星期后才能阅读到(由于他们遵循我的cache 头—"几个星期内都不用检查这个供稿")。另一方面，如果供稿没有改变我也不希望客户端每隔1小时就来检查一下!

`304: Not Modified` 的意思是 “不同的日子，同样的数据(same shit, different day)。”

HTTP 对于这个问题也有一个解决方案。当你第一次请求数据时，服务器返回一个 `Last-Modified` 头。 顾名思义：数据最后修改的时间。 `diveintomark.org` 引用的这张背景图片包含一个 `Last-Modified` 头。

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
<mark>Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT</mark>
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

如果第二(第三，第四）次请求同样一个资源，你可以在你的请求中发送一个 `If-Modified-Since` 头，其值为你上次从服务器返回的时间。如果从那时开始，数据已经发成过变化，服务器会忽略 `If-Modified-Since` 头并返回新数据和 `200` 状态码给你。否则的话，服务器将发回一个特殊的HTTP `304` 状态码, 它的含义是“从上次请求到现在数据没有发生过变化.” 你可以在命令行上使用curl来测试:

```
you@localhost:~$ curl -I <mark>-H "If-Modified-Since: Fri, 22 Aug 2008 04:28:16 GMT"</mar
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

为什么这是一个进步？因为服务器发送 `304` 时，它没有重新发送数据。你得到的仅仅是状态码。即使你的缓存副本已经过期，最后修改时间检查保证你不会在数据没有变化的情况下重新下载它。（额外的好处是，这个 `304` 响应同样也包含了缓存头。代理服务器会在数据已经“过期”的情况下仍然保留数据的副本；希望数据实际上还没有改变，并且下一个请求以 `304` 状态码返回，并更新缓存信息。）

Python的HTTP 库不支持最后修改时间检查，而 `httplib2` 支持。

ETags

ETag 是另一个和最后修改时间检查达到同样目的的方法。使用ETag时，服务器在返回数据的同时在 `ETag` 头里返回一个哈希码(如何生成哈希码完全取决于服务器，唯一的要求是数据改变时哈希码也要改变) `diveintomark.org` 引用的背景图片包含有 `ETag` 头.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
<mark>ETag: "3075-ddc8d800"</mark>
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

`ETag` 的意思是 “太阳底下没有什么新东西。”

当你再次请求同样的数据时，你在 `If-None-Match` 头里放入ETag值。如果数据没有发生改变，服务器将会返回 `304` 状态码。同最后修改时间检查一样，服务器发回的只有 `304` 状态码，不会再一次给你发送同样的数据。通过在请求中包含ETag 哈希码，你告诉服务器如果哈希值匹配就不需要重新发送同样的数据了，因为你仍然保留着上次收到的数据.

再一次使用 `curl`：

```
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

1. ETag 一般使用引号包围，但是引号是值的一部分。它们不是分隔符；`ETag` 头里面唯一的分隔符是 `ETag` 和 `"3075-ddc8d800"` 之间的冒号。这意味着你也需要将引号放在 `If-None-Match` 头发回给服务器。

Python HTTP库不支持ETag，而 `httplib2` 支持.

压缩

当我们谈论HTTP web 服务的时候，你总是会讨论到在线路上来回运送文本数据。可能是XML，也可能是JSON，抑或仅仅是纯文本。不管是什么格式，文本的压缩性能很好。XML章节中的示例供稿在没压缩的情况下是3070 字节，然而在gzip 压缩后只有941 字节。仅仅是原始大小的30%!

HTTP支持若干种压缩算法。最常见的两种是gzip 和 deflate。当你通过HTTP请求资源时，你可以要求服务器以压缩格式返回资源。你在请求中包含一个 `Accept-encoding` 头，里面列出了你支持的压缩算法。如果服务器也支持其中的某一种算法，它就会返回给你压缩后的数据(同时通过 `Content-encoding` 头标识它使用的算法)。接下来的事情就是由你去解压数据了。

Python的 HTTP库不支持压缩，但 `httplib2` 支持。

重定向

好的 URI不会变化，但是有很多URI并没有那么好。网站可能会重新组织，页面移动到新位置。即使是web 服务也可能重新安排。一个联合供稿 `http://example.com/index.xml` 可能会移动到 `http://example.com/xml/atom.xml` 。或者当一个机构扩张和重组的时候，整个域名都可能移动; `http://www.example.com/index.xml` 变成 `http://server-farm-1.example.com/index.xml` .

`Location` 的意思是 “看那边!”

每一次你向HTTP服务器请求资源的时候, 服务器都会在响应中包含一个状态码。 状态码 `200` 的意思是一切正常，这就是你请求的页面; 状态码 `404` 的意思是找不到页面; (你很可能在浏览网页的时候碰到过404)。300 系列的状态码意味着某种形式的重定向。

HTTP 有多种方法表示一个资源已经被移动。最常见两个技术是状态码 `302` 和 `301` 。 状态码 `302` 是一个 临时重定向; 它意味着, 资源被被临时从这里移动走了; (并且临时地址在 `Location` 头里面给出)。状态码 `301` 是永久重定向; 它意味着，资源被永久的移动了; (并且在 `Location` 头里面给出了新的地址)。如果你得到 `302` 状态码和一个新地址, HTTP规范要求你访问新地址来获得你要的资源，但是下次你要访问同样的资源的时候你应该重新尝试旧的地址。但是如果你得到 `301` 状态码和新地址, 你从今以后都应该使用新的地址。

`urllib.request` 模块在从HTTP服务器收到对应的状态码的时候会自动“跟随”重定向, 但它不会告诉你它这么干了。你最后得到了你请求的数据，但是你永远也不会知道下层的库友好的帮助你跟随了重定向。结果是，你继续访问旧的地址，每一次你都会得到新地址的重定向，每一次 `urllib.request` 模块都会友好的帮你跟随重定向。换句话说，它将永久重定向当成临时重定向来处理。这意味着两个来回而不是一个，这对你和服务器都不好。

`httplib2` 帮你处理了永久重定向。它不仅会告诉你发生了永久重定向，而且它会在本地记录这些重定向，并且在发送请求前自动重写为重定向后的URL。

避免通过 HTTP 重复地获取数据

我们来举个例子，你想要通过HTTP下载一个资源, 比如说一个Atom 供稿。作为一个供稿, 你不会只下载一次，你会一次又一次的下载它。 (大部分的供稿阅读器会美一小时检查一次更新。) 让我们先用最粗糙和最快的方法来实现它，接着再来看看怎样改进。

```
>>> import urllib.request
>>> a_url = 'http://diveintopython3.org/examples/feed.xml'

<class 'bytes'>
>>> print(data)
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  …
```

1. 在Python中通过HTTP下载东西是非常简单的; 实际上，只需要一行代码。 `urllib.request` 模块有一个方便的函数 `urlopen()` ，它接受你所要获取的页面地址，然后返回一个类文件对象，您只要调用它的 `read()` 方法就可以获得网页的全部内容。没有比这更简单的了。
2. `urlopen().read()` 方法总是返回 `bytes` 对象,而不是字符串。记住字节仅仅是字节，字符只是一种抽象。 HTTP 服务器不关心抽象的东西。如果你请求一个资源，你得到字节。如果你需要一个字符串，你需要确定字符编码,并显式的将其转化成字符串。

那么，有什么问题呢？作为开发或测试中的快速试验，没有什么不妥的地方。我总是这么干。我需要供稿的内容，然后我拿到了它。相同的技术对任何网页都有效。但一旦你考虑到你需要定期访问Web服务的时候，(例如 每隔1小时请求一下这个供稿), 这样的做法就显得很低效和粗暴了。

线路上是什么？

为了说明为什么这是低效和粗暴的，我们来打开Python的HTTP库的调试功能，看看什么东西被发送到了线路上(即网络上).

```
>>> from http.client import HTTPConnection

>>> from urllib.request import urlopen

Connection: close
reply: 'HTTP/1.1 200 OK'
…further debugging information omitted…
```

1. 正如我在这章开头提到的， `urllib.request` 依赖另一个标准Python库， `http.client` 。正常情况下你不需要直接接触 `http.client` 。 (`urllib.request` 模块会自动导入它。) 我

们在这里导入它是为了让我们能够打开 `HTTPConnection` 类的调试开关， `urllib.request` 使用这个类去连接HTTP服务器。
2. 调式开关已经打开，有关HTTP请求和响应的信息会实时的打印出来。正如你所看见的，当你请求Atom 供稿时， `urllib.request` 模块向服务器发送了5行数据。
3. 第一行指定了你使用的HTTP方法和你访问的资源的路径(不包含域名)。
4. 第二行指定了你请求的供稿所在的域名。
5. 第三行指定客户端支持的压缩算法。我之前提到过， `urllib.request` 默认不支持压缩。
6. 第四行说明了发送请求的库的名字。默认情况下是 `Python-urllib` 加上版本号。 `urllib.request` 和 `httplib2` 都支持更改用户代理, 直接向请求里面加一个 `User-Agent` 头就可以了(默认值会被覆盖).

我们下载了3070字节，但其实我们可以只下载941个字节.

现在让我们来看看服务器返回了什么。

```
# continued from previous example

Server: Apache

Accept-Ranges: bytes

Expires: Mon, 01 Jun 2009 19:23:06 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml

>>> len(data)
3070
```

1. `urllib.request.urlopen()` 函数返回的 `response` 对象包含了服务器返回的所有HTTP头。它也提供了下载实际数据的方法，这个我们等一下讲。
2. 服务器提供了它处理你的请求时的时间。
3. 这个响应包含了 `Last-Modified` 头。
4. 这个响应包含了 `ETag` 头。
5. 数据的长度是3070字节。请注意什么东西没有出现在这里: `Content-encoding` 头。你的请求表示你只接受未压缩的数据，(`Accept-encoding: identity`), 然后当然，响应确实包含未压缩的数据。
6. 这个响应包含缓存头，表明这个供稿可以缓存长达24小时。(86400 秒).
7. 最后，通过调用 `response.read()` 下载实际的数据. 你从 `len()` 函数可以看出，一下子就把整个3070个字节下载下来了。

正如你所看见的，这个代码已经是低效的了；它请求(并接收)了未压缩的数据。我知道服务器实际上是支持gzip 压缩的, 但HTTP 压缩是一个可选项。我们不主动要求，服务器不会执行。这意味这在可以只下载941字节的情况下我们下载了3070个字节。Bad dog, no biscuit.

别急，还有更糟糕的。为了说明这段代码有多么的低效，让我再次请求一下同一个供稿。

```
# continued from the previous example
>>> response2 = urlopen('http://diveintopython3.org/examples/feed.xml')
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
Accept-Encoding: identity
User-Agent: Python-urllib/3.1'
Connection: close
reply: 'HTTP/1.1 200 OK'
…further debugging information omitted…
```

注意到这个请求有什么特别之处吗？它没有变化。它同第一个请求完全一样。没有 `If-Modified-Since` 头. 没有 `If-None-Match` 头. 没有尊重缓存头，也仍然没有压缩。

然后，当你发送同样的请求的时候会发生什么呢？你又一次得到同样的响应。

```
# continued from the previous example

Date: Mon, 01 Jun 2009 03:58:00 GMT
Server: Apache
Last-Modified: Sun, 31 May 2009 22:51:11 GMT
ETag: "bfe-255ef5c0"
Accept-Ranges: bytes
Content-Length: 3070
Cache-Control: max-age=86400
Expires: Tue, 02 Jun 2009 03:58:00 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml
>>> data2 = response2.read()

3070

True
```

1. 服务器仍然在发送同样的聪明的头: `Cache-Control` 和 `Expires` 用于允许缓存, `Last-Modified` 和 `ETag` 用于“是否变化”的跟踪。甚至是 `Vary: Accept-Encoding` 头暗示只要你请求，服务器就能支持压缩。但是你没有。
2. 再一次，获取这个数据下载了一共3070个字节…
3. …和你上一次下载的3070字节完全一致。

HTTP 设计的能比这样工作的更好。 `urllib` 使用HTTP就像我说西班牙语一样 — 可以表达基本的意思，但是不足以保持一个对话。HTTP 是一个对话。是时候更新到一个可以流利的讲HTTP的库了。

介绍 `httplib2`

在你使用 `httplib2` 前, 你需要先安装它。 访问 `code.google.com/p/httplib2/` 并下载最新版本。 `httplib2` 对于Python 2.x 和 Python 3.x都有对应的版本; 请确保你下载的是Python 3 的版本, 名字类似 `httplib2-python3-0.5.0.zip` 。

解压该档案，打开一个终端窗口, 然后切换到刚生成的 `httplib2` 目录。在Windows 上，请打开 `开始` 菜单, 选择 `运行` , 输入 `cmd.exe` 最后按 `回车(ENTER)` .

```
c:\Users\pilgrim\Downloads> <mark>dir</mark>
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\Downloads

07/28/2009  12:36 PM    <DIR>          .
07/28/2009  12:36 PM    <DIR>          ..
07/28/2009  12:36 PM    <DIR>          httplib2-python3-0.5.0
07/28/2009  12:33 PM            18,997 httplib2-python3-0.5.0.zip
               1 File(s)         18,997 bytes
               3 Dir(s)  61,496,684,544 bytes free

c:\Users\pilgrim\Downloads> <mark>cd httplib2-python3-0.5.0</mark>
c:\Users\pilgrim\Downloads\httplib2-python3-0.5.0> <mark>c:\python31\python.exe setup.py
running install
running build
running build_py
running install_lib
creating c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\iri2uri.py -> c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\__init__.py -> c:\python31\Lib\site-packages\httplib2
byte-compiling c:\python31\Lib\site-packages\httplib2\iri2uri.py to iri2uri.pyc
byte-compiling c:\python31\Lib\site-packages\httplib2\__init__.py to __init__.pyc
running install_egg_info
Writing c:\python31\Lib\site-packages\httplib2-python3_0.5.0-py3.1.egg-info
```

在Mac OS X上, 运行位于 `/Applications/Utilities/` 目录下的 `Terminal.app` 程序。在Linux上，运行 `终端(Terminal)` 程序, 该程序一般位于你的 `应用程序` 菜单，在 `Accessories` 或者 `系统(System)` 下面。

```
you@localhost:~/Desktop$ <mark>unzip httplib2-python3-0.5.0.zip</mark>
Archive:  httplib2-python3-0.5.0.zip
  inflating: httplib2-python3-0.5.0/README
  inflating: httplib2-python3-0.5.0/setup.py
  inflating: httplib2-python3-0.5.0/PKG-INFO
  inflating: httplib2-python3-0.5.0/httplib2/__init__.py
  inflating: httplib2-python3-0.5.0/httplib2/iri2uri.py
you@localhost:~/Desktop$ <mark>cd httplib2-python3-0.5.0/</mark>
you@localhost:~/Desktop/httplib2-python3-0.5.0$ <mark>sudo python3 setup.py install</mark
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-3.1
creating build/lib.linux-x86_64-3.1/httplib2
copying httplib2/iri2uri.py -> build/lib.linux-x86_64-3.1/httplib2
copying httplib2/__init__.py -> build/lib.linux-x86_64-3.1/httplib2
running install_lib
creating /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/iri2uri.py -> /usr/local/lib/python3.1/dist-p
copying build/lib.linux-x86_64-3.1/httplib2/__init__.py -> /usr/local/lib/python3.1/dist-
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/iri2uri.py to iri2uri.pyc
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/__init__.py to __init__.py
running install_egg_info
Writing /usr/local/lib/python3.1/dist-packages/httplib2-python3_0.5.0.egg-info
```

要使用 `httplib2` , 请创建一个 `httplib2.Http` 类的实例。

```
>>> import httplib2

200

b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content)
3070
```

1. `httplib2` 的主要接口是 `Http` 对象。你创建 `Http` 对象时总是应该传入一个目录名，具体原因你会在下一节看见。目录不需要事先存在， `httplib2` 会在必要的时候创建它。
2. 一旦你有了 `Http` 对象, 获取数据非常简单，以你要的数据的地址作为参数调用 `request()` 方法就可以了。这会对该URL执行一个HTTP `GET` 请求. (这一章下面你会看见怎样执行其他HTTP 请求, 比如 `POST` 。)
3. `request()` 方法返回两个值。第一个是一个 `httplib2.Response` 对象，其中包含了服务器返回的所有HTTP头。比如, `status` 为 `200` 表示请求成功。
4. `content` 变量包含了HTTP服务器返回的实际数据。数据以 `bytes` 对象返回，不是字符串。 如果你需要一个字符串，你需要确定字符编码并自己进行转换。

> ☞你很可能只需要一个 `httplib2.Http` 对象。当然存在足够的理由来创建多个，但是只有当你清楚创建多个的原因的时候才应该这样做。从不同的URL获取数据不是一个充分的理由，重用 `Http` 对象并调用 `request()` 方法两次就可以了。

关于 `httplib2` 返回字节串而不是字符串的简短解释

字节串。字符串。真麻烦啊。为什么 `httplib2` 不能替你把转换做了呢？由于决定字符编码的规则依赖于你请求的资源的类型，导致自动转化很复杂。 `httplib2` 怎么知道你要请求的资源的类型呢？通常类型会在 `Content-Type` HTTP 头里面列出,但是这是HTTP的可选特性，并且并非所有的HTTP服务器都支持。如果HTTP响应没有包含这个头，那就留给客户端去猜了。(这通常被称为“内容嗅探(content sniffing)”， 但它从来就不是完美的。)

如果你知道你期待的资源是什么类型的(这个例子中是XML文档), 也许你应该直接将返回的 `字节串(bytes)` 对象传给 `xml.etree.ElementTree.parse()` 函数。只要(像这个文档一样)XML文档自己包含字符编码信息，这是可以工作的。但是字符编码信息是一个可选特性并非所有XML文档包含这样的信息。如果一个XML文档不包含编码信息，客户端应该去查看 `Content-Type` HTTP 头, 里面应该包含一个 `charset` 参数。

I support RFC 3023 t-shirt

但问题更糟糕。现在字符编码信息可能在两个地方：在XML文档自己内部，在 `Content-Type` HTTP 头里面。如果信息在两个地方都出现了，哪个优先呢？根据RFC 3023 (我发誓，这不是我编的), 如果在 `Content-Type` HTTP头里面给出的媒体类型(media type)是 `application/xml` , `application/xml-dtd` , `application/xml-external-parsed-entity` , 或者是任何 `application/xml` 的子类型，比如 `application/atom+xml` 或者 `application/rss+xml` 亦或是 `application/rdf+xml` , 那么编码是

1. `Content-Type` HTTP头的 `charset` 参数给出的编码, 或者
2. 文档内的XML声明的 `encoding` 属性给出的编码, 或者
3. UTF-8

相反，如果在 `Content-Type` HTTP头里面给出的媒体类型(media type)是 `text/xml` , `text/xml-external-parsed-entity` , 或者任何 `text/AnythingAtAll+xml` 这样的子类型, 那么文档内的XML声明的 `encoding` 属性完全被忽略，编码是

1. `Content-Type` HTTP头的 `charset` 参数给出的编码, 或者
2. `us-ascii`

而且这还只是针对XML文档的规则。对于HTML文档，网页浏览器创造了用于内容嗅探的复杂规则(byzantine rules for content-sniffing) [PDF], 我们正试图搞清楚它们。 .

"欢迎提交补丁."

`httplib2` 怎样处理缓存。

还记的在前一节我说过你总是应该在创建 `httplib2.Http` 对象是提供一个目录名吗? 缓存就是这样做的目的。

```
# continued from the previous example

200

b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content2)
3070
```

1. 没什么惊奇的东西。跟上次一样，只不过你把结果放入两个新的变量。
2. HTTP `状态(status)` 码同上次一样还是 `200` 。
3. 下载的内容也一样。

谁关心这些东西啊？退出你的Python交互shell 然后打开一个新的会话，我来给你演示。

```
# NOT continued from previous example!
# Please exit out of the interactive shell
# and launch a new one.
>>> import httplib2

3070

200

True
```

1. 让我们打开调试开关来看看线路上是什么。这是使用 `httplib2` 打开 `http.client` 调试开关的方法. `httplib2` 会打印出发给服务器的所有数据以及一些返回的关键信息。
2. 使用同之前一样的目录创建 `httplib2.Http` 对象。

3. 请求同之前一样的URL。 什么也没有发生。 更准确的说，没有东西发送到服务器，没有东西从服务器返回。没有任何形式的网络活动。
4. 但我们还是接收到了数据，实际上是所有的数据。
5. 我们也接收到表示请求成功的HTTP状态码。
6. 这里是奥秘所在: 响应是从 `httplib2` 的本地缓存构造出来的。你创建 `httplib2.Http` 对象是传入的目录里面保存了所有 `httplib2` 执行过的操作的缓存。

线路上有什么？没有东西。

> ☞如果你想要打开 `httplib2` 的调试开关，你需要设置一个模块级的常量 (`httplib2.debuglevel`), 然后再创建 `httplib2.Http` 对象。如果你希望关闭调试，你需要改变同一个模块级常量, 接着创建一个新的 `httplib2.Http` 对象。

你刚刚请求过这个URL的数据。那个请求是成功的(`状态码: 200`)。该响应不仅包含feed数据，也包含一系列缓存头，告诉那些关注着的人这个资源可以缓存长达24小时 (`Cache-Control: max-age=86400` , 24小时所对应的秒数)。 `httplib2` 理解并尊重那些缓存头，并且它会在 `.cache` 目录(你在创建 `Http` 对象时提供的)保存之前的响应。缓存还没有过期，所以你第二次请求该URL的数据时， `httplib2` 不会去访问网络，直接返回缓存着的数据。

我说的很简单，但是很显然在这简单后面隐藏了很多复杂的东西。 `httplib2` 会自动处理HTTP缓存，并且这是默认的行为. 如果由于某些原因你需要知道响应是否来自缓存，你可以检查 `response.fromcache` . 否则的话，它工作的很好。

现在，假设你有数据缓存着，但是你希望跳过缓存并且重新请求远程服务器。浏览器有时候会应用户的要求这么做。比如说，按 `F5` 刷新当前页面，但是按 `Ctrl+F5` 会跳过缓存并向远程服务器重新请求当前页面。你可能会想“嗯，我只要从本地缓存删除数据，然后再次请求就可以了。” 你可以这么干，但是请记住, 不只是你和远程服务器会牵扯其中。那些中继代理服务器呢？ 它们完全不受你的控制，并且它们可能还有那份数据的缓存，然后很高兴的将其返回给你, 因为(对它们来说)缓存仍然是有效的。

你应该使用HTTP的特性来保证你的请求最终到达远程服务器，而不是修改本地缓存然后听天由命。

```
# continued from the previous example
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml',

send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
user-agent: Python-httplib2/$Rev: 259 $
accept-encoding: deflate, gzip
cache-control: no-cache'
reply: 'HTTP/1.1 200 OK'
…further debugging information omitted…
>>> response2.status
200

False

{'status': '200',
 'content-length': '3070',
 'content-location': 'http://diveintopython3.org/examples/feed.xml',
 'accept-ranges': 'bytes',
 'expires': 'Wed, 03 Jun 2009 00:40:26 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Sun, 31 May 2009 22:51:11 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip',
 'etag': '"bfe-255ef5c0"',
 'cache-control': 'max-age=86400',
 'date': 'Tue, 02 Jun 2009 00:40:26 GMT',
 'content-type': 'application/xml'}
```

1. `httplib2` 允许你添加任意的HTTP头部到发出的请求里。为了跳过所有缓存(不仅仅是你本地的磁盘缓存，也包括任何处于你和远程服务器之间的缓存代理服务器), 在 `headers` 字典里面加入 `no-cache` 头就可以了。
2. 现在你可以看见 `httplib2` 初始化了一个网络请求。 `httplib2` 理解并尊重两个方向的缓存头， — 作为接受的响应的一部分以及作为发出的请求的一部分. 它注意到你加入了一个 `no-cache` 头，所以它完全跳过了本地的缓存，然后不得不去访问网络来请求数据。
3. 这个响应不是从本地缓存生成的。你当然知道这一点，因为你看见了发出的请求的调试信息。但是从程序上再验证一下也不错。
4. 请求成功；你再次从远程服务器下载了整个供稿。当然，服务器同供稿数据一起也返回了完整的HTTP头。这里面也包含缓存头，`httplib2` 会使用它来更新它的本地缓存，希望你下次请求该供稿时能够避免网络请求。HTTP缓存被设计为尽量最大化缓存命中率和最小化网络访问。即使你这一次跳过了缓存，服务器仍非常乐意你能缓存结果以备下一次请求

`httplib2` 怎么处理 `Last-Modified` 和 `ETag` 头

`Cache-Control` 和 `Expires` 缓存头 被称为新鲜度指标*(freshness indicators)*。他们毫不含糊告诉缓存，你可以完全避免所有网络访问，直到缓存过期。而这正是你在前一节所看到的: 给出一个新鲜度指标，`httplib2` 不会产生哪怕是一个字节的网络活动 就可以提供缓存了的数据(当然除非你显式的要求跳过缓存).

那如果数据可能已经改变了, 但实际没有呢? HTTP 为这种目的定义了 `Last-Modified` 和 `Etag` 头。 这些头被称为验证器(*validators*)。如果本地缓存已经不是新鲜的，客户端可以在下一个请求的时候发送验证器来检查数据实际上有没有改变。如果数据没有改变，服务器返回 `304` 状态码，但不返回数据。 所以虽然还会在网络上有一个来回，但是你最终可以少下载一点字节。

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')

connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'

{'-content-encoding': 'gzip',
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 <mark>'etag': '"7f806d-1a01-9fb97900"',</mark>
 <mark>'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',</mark>
 'server': 'Apache',
 'status': '200',
 'vary': 'Accept-Encoding,User-Agent'}

6657
```

1. 取代供稿，我们这一次要下载的是网站的主页，是HTML格式的。这是你第一次请求这个页面， `httplib2` 没什么能做的，它在请求中发出最少量的头。
2. 响应包含了多个HTTP头... 但是没有缓存信息。然而，它包含了 `ETag` 和 `Last-Modified` 头。
3. 在我写这个例子的时候，这个页面有6657字节。在那之后，它很可能已经变了, 但是不用担心这一点。

```
# continued from the previous example

connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org

accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'

True

200

'304'

6657
```

1. 你再次请求同一个页面，使用同一个 `Http` 对象(以及同一个本地缓存)。

2. `httplib2` 将 `ETag` validator 通过 `If-None-Match` 头发送回服务器。
3. `httplib2` 也将 `Last-Modified` validator 通过 `If-Modified-Since` 头发送回服务器。
4. 服务器查看这些验证器(validators), 查看你请求的页面，然后判读得出页面在上次请求之后没有改变过, 所以它发回了 `304` 状态码不带数据.
5. 回到客户端，`httplib2` 注意到 `304` 状态码并从它的缓存加载页面的内容。
6. 这可能会让人有些困惑。这里实际上有两个 状态码 — `304` (服务器这次返回的, 导致 `httplib2` 查看它的缓存), 和 `200` (服务器上次返回的, 并和页面数据一起保存在 `httplib2` 的缓存里)。 `response.status` 返回缓存里的那个。
7. 如果你需要服务器返回的原始的状态码，你可以从 `response.dict` 里面找到, 它是包含服务器返回的真实头部的字典.
8. 然而，数据还是保存在了 `content` 变量里。一般来说，你不需要关心为什么响应是从缓存里面来的。(你甚至不需要知道它是从缓存里来的， 这是一件好事。 `httplib2` 足够聪明，允许你傻瓜一点。）`request()` 返回的时候, `httplib2` 就已经更新了缓存并把数据返回给你了。

`http2lib` 怎么处理压缩

“我们两种音乐都有，乡村的和西方的。”

HTTP支持两种类型的压缩。 `httplib2` 都支持。

```
>>> response, content = h.request('http://diveintopython3.org/')
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org

user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
>>> print(dict(response.items()))

 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"',
 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '304',
 'vary': 'Accept-Encoding,User-Agent'}
```

1. 每一次 `httplib2` 发送请求，它包含了 `Accept-Encoding` 头来告诉服务器它能够处理 `deflate` 或者 `gzip` 压缩。
2. 这个例子中，服务器返回了gzip压缩过的负载，当 `request()` 方法返回的时候， `httplib2` 就已经解压缩了响应的体(body)并将其放在 `content` 变量里。如果你想知道响应是否压缩过, 你可以检查 `response['-content-encoding']` ; 否则，不用担心了.

`httplib2` 怎样处理重定向

HTTP 定义了 两种类型的重定向: 临时的和永久的。对于临时重定向，除了跟随它们其他没有什么特别要做的，`httplib2` 会自动处理跟随。

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')

connect: (diveintopython3.org, 80)

Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'

Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
```

1. 这个URL上没有供稿。我设置了服务器让其发出一个到正确地址的临时重定向。
2. 这是请求。
3. 这是响应: `302 Found` 。这里没有显示出来，这个响应也包含由一个 `Location` 头给出实际的URL.
4. `httplib2` 马上转身并跟随重定向，发出另一个到在 `Location` 头里面给出的URL: `http://diveintopython3.org/examples/feed.xml` 的请求。

“跟随” 一个重定向就是这个例子展示的那么多。`httplib2` 发送一个请求到你要求的URL。服务器返回一个响应说“不，不, 看那边.” `httplib2` 给新的URL发送另一个请求.

```
# continued from the previous example

{'status': '200',
 'content-length': '3070',

 'accept-ranges': 'bytes',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Wed, 03 Jun 2009 02:20:15 GMT',
 'connection': 'close',

 'etag': '"bfe-4cbbf5c0"',

 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'application/xml'}
```

1. 你调用 `request()` 方法返回的 `response` 是最终URL的响应。
2. `httplib2` 会将最终的 URL以 `content-location` 加入到 `response` 字典中。这不是服务器返回的头，它特定于 `httplib2` 。
3. 没什么特别的理由, 这个供稿是压缩过的.
4. 并且是可缓存的. (等一下你会看到，这很重要。)

你得到的 `response` 给了你最终 URL的相关信息。如果你希望那些最后重定向到最终URL的中间URL的信息呢？`httplib2` 也能帮你。

```
# continued from the previous example

{'status': '302',
 'content-length': '228',
 'content-location': 'http://diveintopython3.org/examples/feed-302.xml',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'server': 'Apache',
 'connection': 'close',
 'location': 'http://diveintopython3.org/examples/feed.xml',
 'cache-control': 'max-age=86400',
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'text/html; charset=iso-8859-1'}

<class 'httplib2.Response'>
>>> type(response.previous)
<class 'httplib2.Response'>

>>>
```

1. `response.previous` 属性持有前一个响应对象的引用， `httplib2` 跟随那个响应获得了当前的响应对象。
2. `response` 和 `response.previous` 都是 `httplib2.Response` 对象。
3. 这意味着你可以通过 `response.previous.previous` 来反向跟踪重定向链到更前的请求。(场景: 一个URL 重定向到第二个URL，它又重定向到第三个URL。这可能发生!) 在这例子里，我们已经到达了重定向链的开头，所有这个属性是 `None` .

如果我们再次请求同一个URL会发生什么?

```
# continued from the previous example

connect: (diveintopython3.org, 80)

Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'

True
```

1. 同一个URL, 同一个 `httplib2.Http` 对象 (所以也是同一个缓存)。
2. `302` 响应没有缓存，所以 `httplib2` 对同一个 URL发送了另一个请求。
3. 再一次，服务器以 `302` 响应。但是请注意什么没有 发生: 没有第二个到最终URL, `http://diveintopython3.org/examples/feed.xml` 的请求。原因是缓存 (还记的你在前一个例子中看到的 `Cache-Control` 头吗？)。 一旦 `httplib2` 收到 `302 Found` 状态码, 它在发出新的请求前检查它的缓存. 缓存中有 `http://diveintopython3.org/examples/feed.xml` 的一份新鲜副本, 所以不需要重新请求它了。
4. 当 `request()` 方法返回的时候，它已经从缓存中读取了feed数据并返回了它。当然，它和你上次收到的数据是一样的。

换句话说，对于临时重定向你不需要做什么特别的处理。 `httplib2` 会自动跟随它们，而一个URL重定向到另一个这个事实上不会影响 `httplib2` 对压缩，缓存， `ETags` , 或者任何其他HTTP特性的支持。

永久重定向同样也很简单。

```
# continued from the previous example

connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-301.xml HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'

True
```

1. 又一次，这个URL实际上并不存在。我设置我的服务器来执行一个永久重定向到 `http://diveintopython3.org/examples/feed.xml` .
2. 这就是: 状态码 `301` 。 但是再次注意什么没有发生: 没有发送到重定向后的URL的请求。为什么没有? 因为它已经在本地缓存了。
3. `httplib2` “跟随” 重定向到了它的缓存里面。

但是等等! 还有更多!

```
# continued from the previous example

True

True
```

1. 这是临时和永久重定向的区别: 一旦 `httplib2` 跟随了一个永久重定向, 所有后续的对这个URL的请求会被透明的重写到目标URL 而不会接触网络来访问原始的*URL*。 记住, 调试还开着, 但没有任何网络活动的输出。
2. 耶, 响应是从本地缓存获取的。
3. 耶, 你(从缓存里面)得到了整个供稿。

HTTP. 它可以工作。

HTTP GET之外

HTTP web 服务并不限于 `GET` 请求。当你要创建点东西的时候呢？当你在论坛上发表一个评论，更新你的博客，在Twitter 或者 Identi.ca这样的微博客上面发表状态消息的时候, 你很可能已经使用了HTTP `POST` .

Twitter 和 Identi.ca 都提供一个基于HTTP的简单的API来发布并更新你状态(不超过140个字符)。让我们来看看Identi.ca的关于更新状态的API文档 :

> **Identi.ca 的REST API 方法: statuses/update** 更新已认证用户的状态。需要下面格式的 `status` 参数。请求必须是 `POST`.
>
> URL
>
> `https://identi.ca/api/statuses/update._`format`_`
>
> Formats
>
> `xml`, `json`, `rss`, `atom`
>
> HTTP Method(s)
>
> `POST`
>
> Requires Authentication
>
> true
>
> Parameters
>
> `status`. Required. The text of your status update. URL-encode as necessary.

怎么操作呢？要在Identi.ca 发布一条消息, 你需要提交一个HTTP `POST` 请求到 `http://identi.ca/api/statuses/update._format_`. (`format` 字样不是URL的一部分; 你应该将其替换为你希望服务器返回的请求的格式。所以如果需要一个XML格式的返回。你应该向 `https://identi.ca/api/statuses/update.xml` 发送请求。) 请求需要一个参数 `status`, 包含了你的状态更新文本。并且请求必须是已授权的。

授权? 当然。要在Identi.ca上发布你的状态更新, 你得证明你的身份。Identi.ca 不是一个维基; 只有你自己可以更新你的状态。Identi.ca 使用建立在SSL之上的HTTP Basic Authentication (也就是RFC 2617) 来提供安全但方便的认证。 `httplib2` 支持SSL 和 HTTP Basic Authentication, 所以这部分很简单。

`POST` 请求同 `GET` 请求不同, 因为它包含负荷*(payload)*. 负荷是你要发送到服务器的数据。这个API方法必须的参数是 `status`, 并且它应该是*URL*编码过的。 这是一种很简单的序列化格式，将一组键值对(比如字典)转化为一个字符串。

```
'status=Test+update+from+Python+3'
```

1. Python 带有一个工具函数用于URL编码一个字典: `urllib.parse.urlencode()`.
2. 这就是Identi.ca API 所期望的字典。它包含一个键， `status`, 对应值是状态更新文本。
3. 这是URL编码之后的字符串的样子。这就是会通过线路发送到Identi.ca API 服务器的HTTP `POST` 请求中的负荷 .

```
>>> from urllib.parse import urlencode
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> data = {'status': 'Test update from Python 3'}

>>> resp, content = h.request('https://identi.ca/api/statuses/update.xml',
```

1. 这是 `httplib2` 处理认证的方法。 `add_credentials()` 方法记录你的用户名和密码。当 `httplib2` 试图执行请求的时候，服务器会返回一个 `401 Unauthorized` 状态码, 并且列出所有它支持的认证方法(在 `WWW-Authenticate` 头中). `httplib2` 会自动构造 `Authorization` 头并且重新请求该URL.
2. 第二个参数是HTTP请求的类型。这里是 `POST` .
3. 第三个参数是要发送到服务器的负荷 。我们发送包含状态消息的URL编码过的字典。
4. 最后，我们得告诉服务器负荷是URL编码过的数据。

> ☞ `add_credentials()` 方法的第三个参数是该证书有效的域名。你应该总是指定这个参数! 如果你省略了这个参数，并且之后重用这个 `httplib2.Http` 对象访问另一个需要认证的站点，可能会导致 `httplib2` 将一个站点的用户名密码泄漏给其他站点。

发送到线路上的数据:

```
# continued from the previous example
send: b'POST /api/statuses/update.xml HTTP/1.1
Host: identi.ca
Accept-Encoding: identity
Content-Length: 32
content-type: application/x-www-form-urlencoded
user-agent: Python-httplib2/$Rev: 259 $

status=Test+update+from+Python+3'

Host: identi.ca
Accept-Encoding: identity
Content-Length: 32
content-type: application/x-www-form-urlencoded

user-agent: Python-httplib2/$Rev: 259 $

status=Test+update+from+Python+3'
```

1. 第一个请求，服务器以 `401 Unauthorized` 状态码返回。 `httplib2` 从不主动发送认证头，除非服务器明确的要求。这就是服务器要求认证头的方法。
2. `httplib2` 马上转个身，第二次请求同样的URL 。
3. 这一次，包含了你通过 `add_credentials()` 方法加入的用户名和密码。
4. 成功!

请求成功后服务器返回什么？这个完全由web 服务 API决定。 在一些协议里面(就像 Atom Publishing Protocol), 服务器会返回 `201 Created` 状态码，并通过 `Location` 提供新创建的资源的地址。Identi.ca 返回 `200 OK` 和一个包含新创建资源信息的XML 文档。

```
# continued from the previous example

<?xml version="1.0" encoding="UTF-8"?>
<status>

 <truncated>false</truncated>
 <created_at>Wed Jun 10 03:53:46 +0000 2009</created_at>
 <in_reply_to_status_id></in_reply_to_status_id>
 <source>api</source>

 <in_reply_to_user_id></in_reply_to_user_id>
 <in_reply_to_screen_name></in_reply_to_screen_name>
 <favorited>false</favorited>
 <user>
  <id>3212</id>
  <name>Mark Pilgrim</name>
  <screen_name>diveintomark</screen_name>
  <location>27502, US</location>
  <description>tech writer, husband, father</description>
  <profile_image_url>http://avatar.identi.ca/3212-48-20081216000626.png</profile_image_ur
  <url>http://diveintomark.org/</url>
  <protected>false</protected>
  <followers_count>329</followers_count>
  <profile_background_color></profile_background_color>
  <profile_text_color></profile_text_color>
  <profile_link_color></profile_link_color>
  <profile_sidebar_fill_color></profile_sidebar_fill_color>
  <profile_sidebar_border_color></profile_sidebar_border_color>
  <friends_count>2</friends_count>
  <created_at>Wed Jul 02 22:03:58 +0000 2008</created_at>
  <favourites_count>30768</favourites_count>
  <utc_offset>0</utc_offset>
  <time_zone>UTC</time_zone>
  <profile_background_image_url></profile_background_image_url>
  <profile_background_tile>false</profile_background_tile>
  <statuses_count>122</statuses_count>
  <following>false</following>
  <notifications>false</notifications>
</user>
</status>
```

1. 记住，`httplib2` 返回的数据总是字节串(bytes)，不是字符串。为了将其转化为字符串，你需要用合适的字符编码进行解码。Identi.ca的 API总是返回UTF-8编码的结果，所以这部分很简单。
2. 这是我们刚发布的状态消息。
3. 这是新状态消息的唯一标识符。Identi.ca 用这个标识来构造在web上查看该消息的URL。

下面就是这条消息:



HTTP POST之外

HTTP 并不只限于 `GET` 和 `POST` 。 它们当然是最常见的请求类型，特别是在web浏览器里面。 但是web服务API会使用 `GET` 和 `POST` 之外的东西, 对此 `httplib2` 也能处理。

```
# continued from the previous example
>>> from xml.etree import ElementTree as etree

>>> status_id
'5131472'
```

1. 服务器返回的是XML, 对吧? 你知道如何解析XML.
2. `findtext()` 方法找到对应表达式的第一个实例并抽取出它的文本内容。在这个例子中，我们查找 `<id>` 元素.
3. 基于 `<id>` 元素的文本内容，我们可以构造出一个URL用于删除我们刚刚发布的状态消息。
4. 要删除一条消息，你只需要对该URL执行一个HTTP `DELETE` 请求就可以了。

这就是发送到线路上的东西:

```
Host: identi.ca
Accept-Encoding: identity
user-agent: Python-httplib2/$Rev: 259 $

'

Host: identi.ca
Accept-Encoding: identity

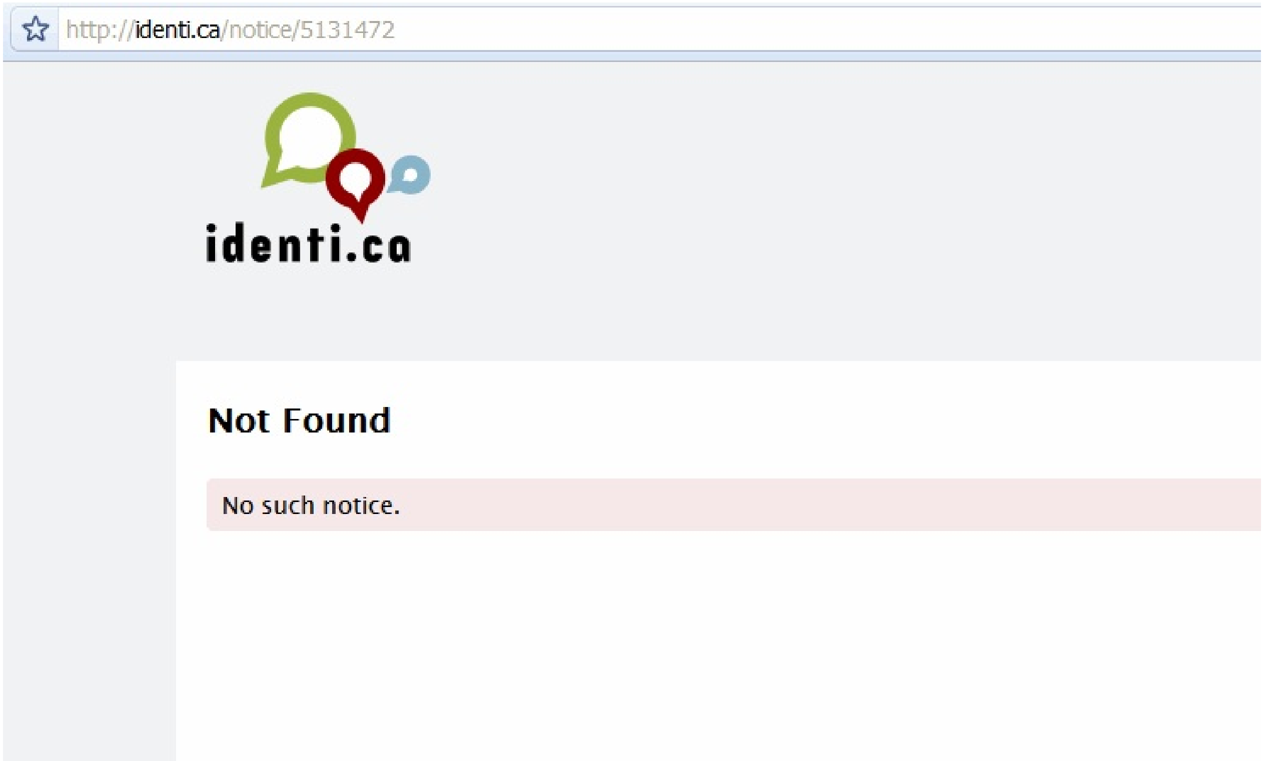
user-agent: Python-httplib2/$Rev: 259 $

'

>>> resp.status
200
```

1. “删除该状态消息.”
2. “对不起，Dave, 恐怕我不能这么干”
3. “没有授权? 恩. 请删除这条消息...
4. ...这是我的用户名和密码。”
5. “应该是完成了!”

证明确实是这样的，它不见了。



进一步阅读

`httplib2`：

- `httplib2` 项目页面

- 更多 `httplib2` 的代码示例
- 正确的处理HTTP缓存: 介绍 `httplib2`
- `httplib2` : HTTP 持久化和认证

HTTP 缓存:

- HTTP 缓存教程 来自 Mark Nottingham
- 怎用使用HTTP头控制缓存 位于 Google Doctype

RFCs:

- RFC 2616: HTTP
- RFC 2617: HTTP Basic Authentication
- RFC 1951: deflate compression
- RFC 1952: gzip compression

Chapter 15 案例研究:将 `chardet` 移植到Python 3

> " Words, words. They're all we have to go on. " — Rosencrantz and Guildenstern are Dead

概述

未知的或者不正确的字符编码是因特网上无效数据(gibberish text)的头号起因。在第 3 章，我们讨论过字符编码的历史，还有Unicode的产生，“一个能处理所有情况的大块头。”如果在网络上不再存在乱码这回事，我会爱上她的…因为所有的编辑系统(authoring system)保存有精确的编码信息，所有的传输协议都支持Unicode，所有处理文本的系统在执行编码间转换的时候都可以保持高度精确。

我也会喜欢pony。

Unicode　pony。

Unipony也行。

这一章我会处理编码的自动检测。

什么是字符编码自动检测？

它是指当面对一串不知道编码信息的字节流的时候，尝试着确定一种编码方式以使我们能够读懂其中的文本内容。它就像我们没有解密钥匙的时候，尝试破解出编码。

那不是不可能的吗？

通常来说，是的，不可能。但是，有一些编码方式为特定的语言做了优化，而语言并非随机存在的。有一些字符序列在某种语言中总是会出现，而其他一些序列对该语言来说则毫无意义。一个熟练掌握英语的人翻开报纸，然后发现“txzqJv 2!dasd0a QqdKjvz”这样一些序列，他会马上意识到这不是英语（即使它完全由英语中的字母组成）。通过研究许多具有“代表性(typical)”的文本，计算机算法可以模拟人的这种对语言的感知，并且对一段文本的语言做出启发性的猜测。

换句话说就是，检测编码信息就是检测语言的类型，并辅之一些额外信息，比如每种语言通常会使用哪些编码方式。

这样的算法存在吗？

结果证明，是的，它存在。所有主流的浏览器都有字符编码自动检测的功能，因为因特网上总是充斥着大量缺乏编码信息的页面。Mozilla Firefox包含有一个自动检测字符编码的库，它是开源的。我将它导入到了Python 2，并且取绰号为 `chardet` 模块。这一章中，我会带领你一步一步地将 `chardet` 模块从Python 2移植到Python 3。

介绍 `chardet` 模块

在开始代码移植之前，如果我们能理解代码是如何工作的这将非常有帮助！以下是一个简明地关于 `chardet` 模块代码结构的手册。 `chardet` 库太大，不可能都放在这儿，但是你可以从 `chardet.feedparser.org` 下载它。

编码检测就是语言检测。

`universaldetector.py` 是检测算法的主入口点，它包含一个类，即 `UniversalDetector` 。（可能你会认为入口点是 `chardet/__init__.py` 中的 `detect` 函数，但是它只是一个便捷的包装方法，它会创建 `UniversalDetector` 对象，调用对象的方法，然后返回其结果。）

`UniversalDetector` 共处理 5 类编码方式：

1. 包含字节顺序标记(BOM)的UTF-n。它包括UTF-8，大尾端和小尾端的UTF-16，还有所有 4 字节顺序的UTF-32的变体。
2. 转义编码，它们与 7 字节的ASCII编码兼容，非ASCII编码的字符会以一个转义序列打头。比如：ISO-2022-JP(日文)和HZ-GB-2312(中文).
3. 多字节编码，在这种编码方式中，每个字符使用可变长度的字节表示。比如：Big5(中文)，SHIFT_JIS(日文)，EUC-KR(韩文)和缺少BOM标记的UTF-8。
4. 单字节编码，这种编码方式中，每个字符使用一个字节编码。例如：KOI8-R(俄语)，windows-1255(希伯来语)和TIS-620(泰国语)。
5. windows-1252，它主要被根本不知道字符编码的中层管理人员(middle manager)在Microsoft Windows上使用。

有BOM标记的UTF-n

如果文本以BOM标记打头，我们可以合理地假设它使用了UTF-8，UTF-16或者UTF-32编码。（BOM会告诉我们是其中哪一种，这就是它的功能。）这个过程在 `UniversalDetector` 中完成，并且不需要深入处理，会非常快地返回其结果。

转义编码

如果文本包含有可识别的能指示出某种转义编码的转义序列， `UniversalDetector` 会创建一个 `EscCharSetProber` 对象（在 `escprober.py` 中定义），然后以该文本调用它。

`EscCharSetProber` 会根据HZ-GB-2312，ISO-2022-CN，ISO-2022-JP，和ISO-2022-KR(在 `escsm.py` 中定义)来创建一系列的状态机(state machine)。 `EscCharSetProber` 将文本一次一个字节地输入到这些状态机中。如果某一个状态机最终唯一地确定了字符编码， `EscCharSetProber` 迅速地将该有效结果返回给 `UniversalDetector` ，然后 `UniversalDetector` 将其返回给调用者。如果某一状态机进入了非法序列，它会被放弃，然后使用其他的状态机继续处理。

多字节编码

假设没有BOM标记， `UniversalDetector` 会检测该文本是否包含任何高位字符(high-bit character)。如果有的话，它会创建一系列的“探测器(probers)”，检测这段广西是否使用多字节编码，单字节编码，或者作为最后的手段，是否为 `windows-1252` 编码。

这里的多字节编码探测器，即 `MBCSGroupProber` （在 `mbcsgroupprober.py` 中定义），实际上是一个管理一组其他探测器的shell，它用来处理每种多字节编码：Big5，GB2312，EUC-TW，EUC-KR，EUC-JP，SHIFT_JIS和UTF-8。 `MBCSGroupProber` 将文本作为每一个特定编码探测器的输入，并且检测其结果。如果某个探测器报告说它发现了一个非法的字节序列，那么该探测器则会被放弃，不再进一步处理（因此，换句话说就是，任何对 `UniversalDetector` . `feed()` 的子调用都会忽略那个探测器）。如果某一探测器报告说它有足够理由确信找到了正确的字符编码，那么 `MBCSGroupProber` 会将这个好消息传递给 `UniversalDetector` ，然后 `UniversalDetector` 将结果返回给调用者。

大多数的多字节编码探测器从类 `MultiByteCharSetProber` (定义在 `mbcharsetprober.py` 中)继承而来，简单地挂上合适的状态机和分布分析器(distribution analyzer)，然后让 `MultiByteCharSetProber` 做剩余的工作。 `MultiByteCharSetProber` 将文本作为特定编码状态机的输入，每次一个字节，寻找能够指示出一个确定的正面或者负面结果的字节序列。同时， `MultiByteCharSetProber` 会将文本作为特定编码分布分析机的输入。

分布分析机（在 `chardistribution.py` 中定义）使用特定语言的模型，此模型中的字符在该语言被使用得最频繁。一旦 `MultiByteCharSetProber` 把足够的文本给了分布分析机，它会根据其中频繁使用字符的数目，字符的总数和特定语言的分配比(distribution ratio)，来计算置信度(confidence rating)。如果置信度足够高， `MultiByteCharSetProber` 会将结果返回给 `MBCSGroupProber` ，然后由 `MBCSGroupProber` 返回给 `UniversalDetector` ，最后 `UniversalDetector` 将其返回给调用者。

对于日语来说检测会更加困难。单字符的分布分析并不总能区别出 `EUC-JP` 和 `SHIFT_JIS` ，所以 `SJISProber` （在 `sjisprober.py` 中定义）也使用双字符的分布分析。 `SJISContextAnalysis` 和 `EUCJPContextAnalysis` (都定义在 `jpcntx.py` 中，并且都从类 `JapaneseContextAnalysis` 中继承）检测文本中的平假名音节字符(Hiragana syllabary characher)的出现次数。一旦处理了足够量的文本，它会返回一个置信度给 `SJISProber` ， `SJISProber` 检查两个分析器的结果，然后将置信度高的那个返回给 `MBCSGroupProber` 。

单字节编码

说正经的，我的Unicode pony哪儿去了？

单字节编码的探测器，即 `SBCSGroupProber` （定义在 `sbcsgroupprober.py` 中），也是一个管理一组其他探测器的shell，它会尝试单字节编码和语言的每种组合： `windows-1251` ， `KOI8-R` ， `ISO-8859-5` ， `MacCyrillic` ， `IBM855` ，and `IBM866` (俄语)； `ISO-8859-7` 和 `windows-1253` (希腊语)； `ISO-8859-5` 和 `windows-1251` (保加利亚语)； `ISO-8859-2` 和 `windows-1250` (匈牙利语)； `TIS-620` (泰国语)； `windows-1255` 和 `ISO-8859-8` (希伯来语)。

`SBCSGroupProber` 将文本输入给这些特定编码+语言的探测器，然后检测它们的返回值。这些探测器的实现为某一个类，即 `SingleByteCharSetProber` (在 `sbcharsetprober.py` 中定义)，它使用语言模型(language model)作为其参数。语言模型定义了典型文本中不同双字符序列出现的频度。 `SingleByteCharSetProber` 处理文本，统计出使用得最频繁的双字符序列。一旦处理了足够多的文本，它会根据频繁使用的序列的数目，字符总数和特定语言的分布系数来计算其置信度。

希伯来语被作为一种特殊的情况处理。如果在双字符分布分析中，文本被认定为是希伯来语， `HebrewProber` (在 `hebrewprober.py` 中定义)会尝试将其从Visual Hebrew（源文本一行一行地被“反向”存储，然后一字不差地显示出来，这样就能从右到左的阅读）和Logical Hebrew（源文本以阅读的顺序保存，在客户端从右到左进行渲染）区别开来。因为有一些字符在两种希伯来语中会以不同的方式编码，这依赖于它们是出现在单词的中间或者末尾，这样我们可以合理的猜测源文本的存储方向，然后返回合适的编码方式(`windows-1255` 对应Logical Hebrew，或者 `ISO-8859-8` 对应Visual Hebrew)。

`windows-1252`

如果 `UniversalDetector` 在文本中检测到一个高位字符，但是其他的多字节编码探测器或者单字节编码探测器都没有返回一个足够可靠的结果，它就会创建一个 `Latin1Prober` 对象(在 `latin1prober.py` 中定义)，尝试从中检测以 `windows-1252` 方式编码的英文文本。这种检测存在其固有的不可靠性，因为在不同的编码中，英文字符通常使用了相同的编码方式。唯一一种区别能出 `windows-1252` 的方法是通过检测常用的符号，比如弯引号(smart quotes)，撇号(curly apostrophes)，版权符号(copyright symbol)等这一类的符号。如果可能 `Latin1Prober` 会自动降低其置信度以使其他更精确的探测器检出结果。

运行 `2to3`

我们将要开始移植 `chardet` 模块到Python 3了。Python 3自带了一个叫做 `2to3` 的实用脚本，它使用Python 2的源代码作为输入，然后尽其可能地将其转换到Python 3的规范。某些情况下这很简单 — 一个被重命名或者被移动到其他模块中的函数 — 但是有些情况下，这个过程会

变得非常复杂。想要了解所有它能做的事情，请参考附录，使用 `2to3` 将代码移植到Python 3。接下来，我们会首先运行一次 `2to3` ，将它作用在 `chardet` 模块上，但是就如你即将看到的，在该自动化工具完成它的魔法表演后，仍然存在许多工作需要我们来收拾。

`chardet` 包被分割为一些不同的文件，它们都放在同一个目录下。 `2to3` 能够立即处理多个文件：只需要将目录名作为命令行参数传递给 `2to3` ，然后它会轮流处理每个文件。

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w chardet\
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- chardet\__init__.py (original)
+++ chardet\__init__.py (refactored)
@@ -18,7 +18,7 @@
 __version__ = "1.0.1"

 def detect(aBuf):
~~-    import universaldetector~~
<ins>+    from . import universaldetector</ins>
     u = universaldetector.UniversalDetector()
     u.reset()
     u.feed(aBuf)
--- chardet\big5prober.py (original)
+++ chardet\big5prober.py (refactored)
@@ -25,10 +25,10 @@
 # 02110-1301  USA
 ######################### END LICENSE BLOCK #########################

~~-from mbcharsetprober import MultiByteCharSetProber~~
~~-from codingstatemachine import CodingStateMachine~~
~~-from chardistribution import Big5DistributionAnalysis~~
~~-from mbcssm import Big5SMModel~~
<ins>+from .mbcharsetprober import MultiByteCharSetProber</ins>
<ins>+from .codingstatemachine import CodingStateMachine</ins>
<ins>+from .chardistribution import Big5DistributionAnalysis</ins>
<ins>+from .mbcssm import Big5SMModel</ins>

 class Big5Prober(MultiByteCharSetProber):
     def __init__(self):
--- chardet\chardistribution.py (original)
+++ chardet\chardistribution.py (refactored)
@@ -25,12 +25,12 @@
 # 02110-1301  USA
 ######################### END LICENSE BLOCK #########################

~~-import constants~~
~~-from euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE, EUCTW_TYPICAL_DISTRIBUTI
~~-from euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE, EUCKR_TYPICAL_DISTRIBUTI
~~-from gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE, GB2312_TYPICAL_DISTRI
~~-from big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE, BIG5_TYPICAL_DISTRIBUTION_R
~~-from jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE, JIS_TYPICAL_DISTRIBUTION_RATIO
<ins>+from . import constants</ins>
<ins>+from .euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE, EUCTW_TYPICAL_DISTRI
<ins>+from .euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE, EUCKR_TYPICAL_DISTRI
<ins>+from .gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE, GB2312_TYPICAL_DI
<ins>+from .big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE, BIG5_TYPICAL_DISTRIBUTI
<ins>+from .jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE, JIS_TYPICAL_DISTRIBUTION_R

 ENOUGH_DATA_THRESHOLD = 1024
 SURE_YES = 0.99
.
.
<mark>. (it goes on like this for a while)</mark>
.
.
RefactoringTool: Files that were modified:
```

```
RefactoringTool: chardet\__init__.py
RefactoringTool: chardet\big5prober.py
RefactoringTool: chardet\chardistribution.py
RefactoringTool: chardet\charsetgroupprober.py
RefactoringTool: chardet\codingstatemachine.py
RefactoringTool: chardet\constants.py
RefactoringTool: chardet\escprober.py
RefactoringTool: chardet\escsm.py
RefactoringTool: chardet\eucjpprober.py
RefactoringTool: chardet\euckrprober.py
RefactoringTool: chardet\euctwprober.py
RefactoringTool: chardet\gb2312prober.py
RefactoringTool: chardet\hebrewprober.py
RefactoringTool: chardet\jpcntx.py
RefactoringTool: chardet\langbulgarianmodel.py
RefactoringTool: chardet\langcyrillicmodel.py
RefactoringTool: chardet\langgreekmodel.py
RefactoringTool: chardet\langhebrewmodel.py
RefactoringTool: chardet\langhungarianmodel.py
RefactoringTool: chardet\langthaimodel.py
RefactoringTool: chardet\latin1prober.py
RefactoringTool: chardet\mbcharsetprober.py
RefactoringTool: chardet\mbcsgroupprober.py
RefactoringTool: chardet\mbcssm.py
RefactoringTool: chardet\sbcharsetprober.py
RefactoringTool: chardet\sbcsgroupprober.py
RefactoringTool: chardet\sjisprober.py
RefactoringTool: chardet\universaldetector.py
RefactoringTool: chardet\utf8prober.py
```

现在我们对测试工具 — `test.py` — 应用 `2to3` 脚本。

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w test.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- test.py (original)
+++ test.py (refactored)
@@ -4,7 +4,7 @@
 count = 0
 u = UniversalDetector()
 for f in glob.glob(sys.argv[1]):
~~-    print f.ljust(60),~~
<ins>+     print(f.ljust(60), end=' ')</ins>
     u.reset()
     for line in file(f, 'rb'):
         u.feed(line)
@@ -12,8 +12,8 @@
     u.close()
     result = u.result
     if result['encoding']:
~~-        print result['encoding'], 'with confidence', result['confidence']~~
<ins>+        print(result['encoding'], 'with confidence', result['confidence'])</ins>
     else:
~~-        print '******** no result'~~
<ins>+        print('******** no result')</ins>
     count += 1
~~-print count, 'tests'~~
<ins>+print(count, 'tests')</ins>
RefactoringTool: Files that were modified:
RefactoringTool: test.py
```

看吧，还不算太难。只是转换了一些impor和print语句。说到这儿，那些import语句原来到底存在什么问题呢？为了回答这个问题，你需要知道 `chardet` 是如果被分割到多个文件的。

题外话，关于多文件模块

`chardet` 是一个多文件模块。我也可以将所有的代码都放在一个文件里(并命名为 `chardet.py`)，但是我没有。我创建了一个目录(叫做 `chardet`)，然后我在那个目录里创建了一个 `__init__.py` 文件。如果*Python*看到目录里有一个 *`__init__.py`* 文件，它会假设该目录里的所有文件都是同一个模块的某部分。模块名为目录的名字。目录中的文件可以引用目录中的其他文件，甚至子目录中的也行。（再讲一分钟这个。）但是整个文件集合被作为一个单独的模块呈现给其他的Python代码 — 就好像所有的函数和类都在一个 `.py` 文件里。

在 `__init__.py` 中到底有些什么？什么也没有。一切。界于两者之间。 `__init__.py` 文件不需要定义任何东西；它确实可以是一个空文件。或者也可以使用它来定义我们的主入口函数。或者把我们所有的函数都放进去。或者其他函数都放，单单不放某一个函数…

> ☞包含有 `__init__.py` 文件的目录总是被看作一个多文件的模块。没有 `__init__.py` 文件的目录中，那些 `.py` 文件是不相关的。

我们来看看它实际上是怎样工作的。

```
>>> import chardet

['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__path__', '__version__', 'detect']

<module 'chardet' from 'C:\Python31\lib\site-packages\chardet\__init__.py'>
```

1. 除了常见的类属性，在 `chardet` 模块中只多了一个 `detect()` 函数。
2. 这是我们发觉 `chardet` 模块不只是一个文件的第一个线索："module"被当作文件 `chardet/` 目录中的 `__init__.py` 文件列出来。

我们再来瞟一眼 `__init__.py` 文件。

```

    u = universaldetector.UniversalDetector()
    u.reset()
    u.feed(aBuf)
    u.close()
    return u.result
```

1. `__init__.py` 文件定义了 `detect()` 函数，它是 `chardet` 库的主入口点。
2. 但是 `detect()` 函数没有任何实际的代码！事实上，它所做的事情只是导入了 `universaldetector` 模块然后开始调用它。但是 `universaldetector` 定义在哪儿？

答案就在那行古怪的 `import` 语句中：

```
from . import universaldetector
```

翻译成中文就是，“导入 `universaldetector` 模块；它跟我在同一目录，”这里的我即指文件 `chardet/__init__.py` 。这是一种提供给多文件模块中文件之间互相引用的方法，不需要担心它会与已经安装的搜索路径中的模块发生命名冲突。该条 `import` 语句只会在 `chardet/` 目录中查找 `universaldetector` 模块。

这两条概念 — `__init__.py` 和相对导入 — 意味着我们可以将模块分割为任意多个块。 `chardet` 模块由36个 `.py` 文件组成 — 36！但我们所需要做的只是使用 `chardet/__init__.py` 文件中定义的某个函数。还有一件事情没有告诉你， `detect()` 使用了相对导入来引用了 `chardet/universaldetector.py` 中定义的一个类，然后这个类又使用了相对导入引用了其他5个文件的内容，它们都在 `chardet/` 目录中。

> ☞如果你发现自己正在用Python写一个大型的库（或者更可能的情况是，当你意识到你的小模块已经变得很大的时候），最好花一些时间将它重构为一个多文件模块。这是Python所擅长的许多事情之一，那就利用一下这个优势吧。

修复 `2to3` 脚本所不能做的

`False` is invalid syntax

你确实有测试样例，对吧？

现在开始真正的测试：使用测试集运行测试工具。由于测试集被设计成可以覆盖所有可能的代码路径，它是用来测试移植后的代码，保证bug不会埋伏在某个地方的一种不错的办法。

```
C:\home\chardet> python test.py tests\*\*
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    from chardet.universaldetector import UniversalDetector
  File "C:\home\chardet\chardet\universaldetector.py", line 51
    self.done = constants.False
                              ^
SyntaxError: invalid syntax
```

唔，一个小麻烦。在Python 3中， `False` 是一个保留字，所以不能把它用作变量名。我们来看一看 `constants.py` 来确定这是在哪儿定义的。以下是 `constants.py` 在执行 `2to3` 脚本之前原来的版本。

```
import __builtin__
if not hasattr(__builtin__, 'False'):
    False = 0
    True = 1
else:
    False = __builtin__.False
    True = __builtin__.True
```

这一段代码用来允许库在低版本的Python 2中运行，在Python 2.3以前，Python没有内置的 `bool` 类型。这段代码检测内置的 `True` 和 `False` 常量是否缺失，如果必要的话则定义它们。

但是，Python 3总是有 `bool` 类型的，所以整个这片代码都没有必要。最简单的方法是将所有的 `constants.True` 和 `constants.False` 都分别替换成 `True` 和 `False` ，然后将这段死代码从 `constants.py` 中移除。

所以 `universaldetector.py` 中的以下行：

```
self.done = constants.False
```

变成了

```
self.done = False
```

啊哈，是不是很有满足感？代码不仅更短了，而且更具可读性。

No module named `constants`

是时候再运行一次 `test.py` 了，看看它能走多远。

```
C:\home\chardet> python test.py tests\*\*
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    from chardet.universaldetector import UniversalDetector
  File "C:\home\chardet\chardet\universaldetector.py", line 29, in <module>
    import constants, sys
ImportError: No module named constants
```

说什么了？不存在叫做 `constants` 的模块？可是当然有 `constants` 这个模块了。它就在 `chardet/constants.py` 中。

还记得什么时候 `2to3` 脚本会修复所有那些导入语句吗？这个包内有许多的相对导入 — 即，在同一个库中，导入其他模块的模块 — 但是在*Python 3*中相对导入的逻辑已经变了。在Python 2中，我们只需要 `import constants` ，然后它就会首先在 `chardet/` 目录中查找。在Python 3中，所有的导入语句默认使用绝对路径。如果想要在Python 3中使用相对导入，你需要显式地说明：

```
from . import constants
```

但是。 `2to3` 脚本难道不是要自动修复这些的吗？好吧，它确实这样做了，但是该条导入语句在同一行组合了两种不同的导入类型：库内部对 `constants` 的相对导入，还有就是对 `sys` 模块的绝对导入， `sys` 模块已经预装在了Python的标准库里。在Python 2里，我们可以将其组

合到一条导入语句中。在Python 3中，我们不能这样做，并且 `2to3` 脚本也不是那样聪明，它不能把这条导入语句分成两条。

解决的办法是把这条导入语句手动的分成两条。所以这条二合一的导入语句：

```
import constants, sys
```

需要变成两条分享的导入语句：

```
from . import constants
import sys
```

在 `chardet` 库中还分散着许多这类问题的变体。某些地方它是“ `import constants, sys` ”；其他一些地方则是“ `import constants, re` ”。修改的方法是一样的：手工地将其分割为两条语句，一条为相对导入准备，另一条用于绝对导入。

前进！

Name `'file'` is not defined

open()代替了原来的file()。PapayaWhip则替代了原来的black

再来一次，运行 `test.py` 来执行我们的测试样例…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    for line in file(f, 'rb'):
NameError: name 'file' is not defined
```

这一条也出乎我的意外，因为在记忆中我一直都在使用这种风格的代码。在Python 2里，全局的 `file()` 函数是 `open()` 函数的一个别名， `open()` 函数是打开文件用于读取的标准方法。在Python 3中，全局的 `file()` 函数不再存在了，但是 `open()` 还保留着。

这样的话，最简单的解决办法就是将 `file()` 调用替换为对 `open()` 的调用：

```
for line in open(f, 'rb'):
```

这即是我关于这个问题想要说的。

Can’t use a string pattern on a bytes-like object

现在事情开始变得有趣了。对于“有趣，”我的意思是“跟地狱一样让人迷茫。”

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 98, in feed
    if self._highBitDetector.search(aBuf):
TypeError: can't use a string pattern on a bytes-like object
```

我们先来看看 `self._highBitDetector` 是什么，然后再来调试这个错误。它被定义在 `UniversalDetector` 类的 `__init__` 方法中。

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(r'[\x80-\xFF]')
```

这段代码预编译一条正则表达式，它用来查找在128–255 (0x80–0xFF)范围内的非ASCII字符。等一下，这似乎不太准确；我需要对更精确的术语来描述它。这个模式用来在128-255范围内查找非ASCII的*bytes*。

问题就出在这儿了。

在Python 2中，字符串是一个字节数组，它的字符编码信息被分开记录着。如果想要Python 2跟踪字符编码，你得使用Unicode编码的字符串(`u''`)。但是在Python 3中，字符串永远都是Python 2中所谓的Unicode编码的字符串 — 即，Unicode字符数组（可能存在可变长字节）。由于这条正则表达式是使用字符串模式定义的，所以它只能用来搜索字符串 — 再强调一次，字符数组。但是我们所搜索的并非字符串，它是一个字节数组。看一看traceback，该错误发生在 `universaldetector.py` ：

```
def feed(self, aBuf):
    .
    .
    .
    if self._mInputState == ePureAscii:
        if self._highBitDetector.search(aBuf):
```

`aBuf` 是什么？让我们原路回到调用 `UniversalDetector.feed()` 的地方。有一处地方调用了它，是测试工具， `test.py` 。

```
u = UniversalDetector()
.
.
.
for line in open(f, 'rb'):
    u.feed(line)
```

非字符数组，而是一个字节数组。

在此处我们找到了答案：`UniversalDetector.feed()` 方法中，`aBuf` 是从磁盘文件中读到的一行。仔细看一看用来打开文件的参数：`'rb'`。`'r'` 是用来读取的；OK，没什么了不起的，我们在读取文件。啊，但是 `'b'` 是用以读取“二进制”数据的。如果没有标记 `'b'`，`for` 循环会一行一行地读取文件，然后将其转换为一个字符串 — Unicode编码的字符数组 — 根据系统默认的编码方式。但是使用 `'b'` 标记后，`for` 循环一行一行地读取文件，然后将其按原样存储为字节数组。该字节数组被传递给了 `UniversalDetector.feed()` 方法，最后给了预编译好的正则表达式，`self._highBitDetector`，用来搜索高位...字符。但是没有字符；有的只是字节。苍天哪。

我们需要该正则表达式搜索的并不是字符数组，而是一个字节数组。

只要我们认识到了这一点，解决办法就有了。使用字符串定义的正则表达式可以搜索字符串。使用字节数组定义的正则表达式可以搜索字节数组。我们只需要改变用来定义正则表达式的参数的类型为字节数组，就可以定义一个字节数组模式。（还有另外一个该问题的实例，在下一行。）

```
 class UniversalDetector:
     def __init__(self):
~~-          self._highBitDetector = re.compile(r'[\x80-\xFF]')~~
~~-          self._escDetector = re.compile(r'(\033|~{)')~~
<ins>+          self._highBitDetector = re.compile(b'[\x80-\xFF]')</ins>
<ins>+          self._escDetector = re.compile(b'(\033|~{)')</ins>
         self._mEscCharSetProber = None
         self._mCharSetProbers = []
         self.reset()
```

在整个代码库内搜索对 `re` 模块的使用发现了另外两个该类型问题的实例，出现在 `charsetprober.py` 文件中。再次，以上代码将正则表达式定义为字符串，但是却将它们作用在 `aBuf` 上，而 `aBuf` 是一个字节数组。解决方案还是一样的：将正则表达式模式定义为字节数组。

```
 class CharSetProber:
     .
     .
     .
     def filter_high_bit_only(self, aBuf):
~~-          aBuf = re.sub(r'([\x00-\x7F])+', ' ', aBuf)~~
<ins>+          aBuf = re.sub(b'([\x00-\x7F])+', b' ', aBuf)</ins>
         return aBuf

     def filter_without_english_letters(self, aBuf):
~~-          aBuf = re.sub(r'([A-Za-z])+', ' ', aBuf)~~
<ins>+          aBuf = re.sub(b'([A-Za-z])+', b' ', aBuf)</ins>
         return aBuf
```

Can't convert `'bytes'` object to `str` implicitly

奇怪，越来越不寻常了...

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 100, in feed
    elif (self._mInputState == ePureAscii) and self._escDetector.search(self._mLastChar +
TypeError: Can't convert 'bytes' object to str implicitly
```

在此存在一个Python解释器与代码风格之间的不协调。 `TypeError` 可以出现在那一行的任意地方，但是traceback不能明确定地指出错误的位置。可能是第一个或者第二个条件语句(conditional)，对traceback来说，它们是一样的。为了缩小调试的范围，我们需要把这条代码分割成两行，像这样：

```
elif (self._mInputState == ePureAscii) and \
    self._escDetector.search(self._mLastChar + aBuf):
```

然后再运行测试工具：

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly
```

啊哈！错误不在第一个条件语句上(`self._mInputState == ePureAscii`)，是第二个的问题。但是，是什么引发了 `TypeError` 错误呢？也许你会想 `search()` 方法需要另外一种类型的参数，但是那样的话，就不会产生当前这种traceback了。Python函数可以使用任何类型参数；只要传递了正确数目的参数，函数就可以执行。如果我们给函数传递了类型不匹配的参数，代码可能就会崩溃，但是这样一来，traceback就会指向函数内部的某一代码块了。但是当前得到的traceback告诉我们，错误就出现在开始调用 `search()` 函数那儿。所以错误肯定就出在 `+` 操作符上，该操作用于构建最终会传递给 `search()` 方法的参数。

从前一次调试的过程中，我们已经知道 `aBuf` 是一个字节数组。那么 `self._mLastChar` 又是什么呢？它是一个在 `reset()` 中定义的实例变量，而 `reset()` 方法刚好就是被 `__init__()` 调用的。

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(b'[\x80-\xFF]')
        self._escDetector = re.compile(b'(\033|~{)')
        self._mEscCharSetProber = None
        self._mCharSetProbers = []
 <mark>self.reset()</mark>

    def reset(self):
        self.result = {'encoding': None, 'confidence': 0.0}
        self.done = False
        self._mStart = True
        self._mGotData = False
        self._mInputState = ePureAscii
 <mark>self._mLastChar = ''</mark>
```

现在我们找到问题的症结所在了。你发现了吗？ `self._mLastChar` 是一个字符串，而 `aBuf` 是一个字节数组。而我们不允许对字符串和字节数组做连接操作 — 即使是空串也不行。

那么， `self._mLastChar` 到底是什么呢？在 `feed()` 方法中，在traceback报告的位置以下几行就是了。

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

<mark>self._mLastChar = aBuf[-1]</mark>
```

`feed()` 方法被一次一次地调用，每次都传递给它几个字节。该方法处理好它收到的字节（以 `aBuf` 传递进去的），然后将最后一个字节保存在 `self._mLastChar` 中，以便下次调用时还会用到。（在多字节编码中， `feed()` 在调用的时候可能只收到了某个字符的一半，然后下次调用时另一半才被传到。）但是因为 `aBuf` 已经变成了一个字节数组，所以 `self._mLastChar` 也需要与其匹配。可以这样做：

```
 def reset(self):
      .
      .
      .
~~-     self._mLastChar = ''~~
<ins>+     self._mLastChar = b''</ins>
```

在代码库中搜索“ `mLastChar` ”， `mbcharsetprober.py` 中也发现一个相似的问题，与之前不同的是，它记录的是最后2个字符。 `MultiByteCharSetProber` 类使用一个单字符列表来记录末尾的两个字符。在Python 3中，这需要使用一个整数列表，因为实际上它记录的并不是是字符，而是字节对象。（字节对象即范围在 `0-255` 内的整数。）

```
 class MultiByteCharSetProber(CharSetProber):
      def __init__(self):
          CharSetProber.__init__(self)
          self._mDistributionAnalyzer = None
          self._mCodingSM = None
~~-         self._mLastChar = ['\x00', '\x00']~~
<ins>+         self._mLastChar = [0, 0]</ins>

      def reset(self):
          CharSetProber.reset(self)
          if self._mCodingSM:
              self._mCodingSM.reset()
          if self._mDistributionAnalyzer:
              self._mDistributionAnalyzer.reset()
~~-         self._mLastChar = ['\x00', '\x00']~~
<ins>+         self._mLastChar = [0, 0]</ins>
```

Unsupported operand type(s) for +: `'int'` and `'bytes'`

有好消息，也有坏消息。好消息是我们一直在前进着…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'
```

…坏消息是，我们好像一直都在原地踏步。

但我们确实一直在取得进展！真的！即使traceback在相同的地方再次出现，这一次的错误毕竟与上次不同。前进！那么，这次又是什么错误呢？上一次我们确认过了，这一行代码不应该会再做连接 `int` 型和字节数组(`bytes`)的操作。事实上，我们刚刚花了相当长一段时间来保证 `self._mLastChar` 是一个字节数组。它怎么会变成 `int` 呢？

答案不在上几行代码中，而在以下几行。

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

<mark>self._mLastChar = aBuf[-1]</mark>
```

字符串中的元素仍然是字符串，字节数组中的元素则为整数。

该错误没有发生在 `feed()` 方法第一次被调用的时候；而是在第二次调用的过程中，在 `self._mLastChar` 被赋值为 `aBuf` 末尾的那个字节之后。好吧，这又会有什么问题呢？因为获取字节数组中的单个元素会产生一个整数，而不是字节数组。它们之间的区别，请看以下在交互式shell中的操作：

```
>>> len(aBuf)
3
>>> mLastChar = aBuf[-1]

191

<class 'int'>

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'

>>> mLastChar
b'\xbf'

b'\xbf\xef\xbb\xbf'
```

1. 定义一个长度为3的字节数组。
2. 字节数组的最后一个元素为191。
3. 它是一个整数。
4. 连接整数和字节数组的操作是不允许的。我们重复了在 `universaldetector.py` 中发现的那个错误。
5. 啊，这就是解决办法了。使用列表分片从数组的最后一个元素中创建一个新的字节数组，而不是直接获取这个元素。即，从最后一个元素开始切割，直到到达数组的末尾。当前 `mLastChar` 是一个长度为1的字节数组。
6. 连接长度分别为1和3的字节数组，则会返回一个新的长度为4的字节数组。

所以，为了保证 `universaldetector.py` 中的 `feed()` 方法不管被调用多少次都能够正常运行，我们需要将 `self._mLastChar` 实例化为一个长度为0的字节数组，并且保证它一直是一个字节数组。

```
 self._escDetector.search(self._mLastChar + aBuf):
          self._mInputState = eEscAscii

~~- self._mLastChar = aBuf[-1]~~
<ins>+ self._mLastChar = aBuf[-1:]</ins>
```

`ord()` expected string of length 1, but `int` found

困了吗？就要完成了…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\utf8prober.py", line 53, in feed
    codingState = self._mCodingSM.next_state(c)
  File "C:\home\chardet\chardet\codingstatemachine.py", line 43, in next_state
    byteCls = self._mModel['classTable'][ord(c)]
TypeError: ord() expected string of length 1, but int found
```

OK，因为 `c` 是 `int` 类型的，但是 `ord()` 需要一个长度为1的字符串。就是这样了。 `c` 在哪儿定义的？

```
# codingstatemachine.py
def next_state(self, c):
    # for each byte we get its class
    # if it is first byte, we also get byte length
    byteCls = self._mModel['classTable'][ord(c)]
```

不是这儿；此处 `c` 只是被传递给了 `next_state()` 函数。我们再上一级看看。

```
# utf8prober.py
def feed(self, aBuf):
    for c in aBuf:
        codingState = self._mCodingSM.next_state(c)
```

看到了吗？在Python 2中， `aBuf` 是一个字符串，所以 `c` 就是一个长度为1的字符串。（那就是我们通过遍历字符串所得到的 — 所有的字符，一次一个。）因为现在 `aBuf` 是一个字节数组，所以 `c` 变成了 `int` 类型的，而不再是长度为1的字符串。也就是说，没有必要再调用 `ord()` 函数了，因为 `c` 已经是 `int` 了！

这样修改：

```
  def next_state(self, c):
       # for each byte we get its class
       # if it is first byte, we also get byte length
~~-      byteCls = self._mModel['classTable'][ord(c)]~~
<ins>+     byteCls = self._mModel['classTable'][c]</ins>
```

在代码库中搜索“ `ord(c)` ”后，发现 `sbcharsetprober.py` 中也有相似的问题…

```
# sbcharsetprober.py
def feed(self, aBuf):
    if not self._mModel['keepEnglishLetter']:
        aBuf = self.filter_without_english_letters(aBuf)
    aLen = len(aBuf)
    if not aLen:
        return self.get_state()
    for c in aBuf:
 <mark>order = self._mModel['charToOrderMap'][ord(c)]</mark>
```

...还有 `latin1prober.py` ...

```
# latin1prober.py
def feed(self, aBuf):
    aBuf = self.filter_with_english_letters(aBuf)
    for c in aBuf:
 <mark>charClass = Latin1_CharToClass[ord(c)]</mark>
```

`c` 在 `aBuf` 中遍历，这就意味着它是一个整数，而非字符串。解决方案是相同的：把 `ord(c)` 就替换成 `c` 。

```
 # sbcharsetprober.py
  def feed(self, aBuf):
      if not self._mModel['keepEnglishLetter']:
          aBuf = self.filter_without_english_letters(aBuf)
      aLen = len(aBuf)
      if not aLen:
          return self.get_state()
      for c in aBuf:
~~-           order = self._mModel['charToOrderMap'][ord(c)]~~
<ins>+           order = self._mModel['charToOrderMap'][c]</ins>

  # latin1prober.py
  def feed(self, aBuf):
      aBuf = self.filter_with_english_letters(aBuf)
      for c in aBuf:
~~-           charClass = Latin1_CharToClass[ord(c)]~~
<ins>+           charClass = Latin1_CharToClass[c]</ins>
```

Unorderable types: `int()` >= `str()`

继续我们的路吧。

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\sjisprober.py", line 68, in feed
    self._mContextAnalyzer.feed(self._mLastChar[2 - charLen :], charLen)
  File "C:\home\chardet\chardet\jpcntx.py", line 145, in feed
    order, charLen = self.get_order(aBuf[i:i+2])
  File "C:\home\chardet\chardet\jpcntx.py", line 176, in get_order
    if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
TypeError: unorderable types: int() >= str()
```

这都是些什么？“Unorderable types”？字节数组与字符串之间的差异引起的问题再一次出现了。看一看以下代码：

```
class SJISContextAnalysis(JapaneseContextAnalysis):
    def get_order(self, aStr):
        if not aStr: return -1, 1
        # find out current char's byte length
 <mark>if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \</mark>
           ((aStr[0] >= '\xE0') and (aStr[0] <= '\xFC')):
            charLen = 2
        else:
            charLen = 1
```

`aStr` 从何而来？再深入栈内看一看：

```
def feed(self, aBuf, aLen):
    .
    .
    .
    i = self._mNeedToSkipCharNum
    while i < aLen:
 <mark>order, charLen = self.get_order(aBuf[i:i+2])</mark>
```

看，是 `aBuf`，我们的老战友。从我们在这一章中所遇到的问题你也可以猜到了问题的关键了，因为 `aBuf` 是一个字节数组。此处 `feed()` 方法并不是整个地将它传递出去；而是先对它执行分片操作。就如你在这章前面看到的，对字节数组执行分片操作的返回值仍然为字节数组，所以传递给 `get_order()` 方法的 `aStr` 仍然是字节数组。

那么以下代码是怎样处理 `aStr` 的呢？它将该字节第一个元素与长度为1的字符串进行比较操作。在Python 2，这是可以的，因为 `aStr` 和 `aBuf` 都是字符串，所以 `aStr[0]` 也是字符串，并且我们允许比较两个字符串的是否相等。但是在Python 3中，`aStr` 和 `aBuf` 都是字节数组，而 `aStr[0]` 就成了一个整数，没有执行显式地强制转换的话，是不能对整数和字符串执行相等性比较的。

在当前情况下，没有必要添加强制转换，这会让代码变得更加复杂。`aStr[0]` 产生一个整数；而我们所比较的对象都是常量(constant)。那就把长度为1的字符串换成整数吧。我们也顺便把 `aStr` 换成 `aBuf` 吧，因为 `aStr` 本来也不是一个字符串。

```
 class SJISContextAnalysis(JapaneseContextAnalysis):
~~-     def get_order(self, aStr):~~
~~-      if not aStr: return -1, 1~~
<ins>+     def get_order(self, aBuf):</ins>
<ins>+      if not aBuf: return -1, 1</ins>
          # find out current char's byte length
~~-         if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \~~
~~-            ((aBuf[0] >= '\xE0') and (aBuf[0] <= '\xFC')):~~
<ins>+         if ((aBuf[0] >= 0x81) and (aBuf[0] <= 0x9F)) or \</ins>
<ins>+            ((aBuf[0] >= 0xE0) and (aBuf[0] <= 0xFC)):</ins>
              charLen = 2
          else:
              charLen = 1

          # return its order if it is hiragana
~~-      if len(aStr) > 1:~~
~~-             if (aStr[0] == '\202') and \~~
~~-                (aStr[1] >= '\x9F') and \~~
~~-                (aStr[1] <= '\xF1'):~~
~~-                return ord(aStr[1]) - 0x9F, charLen~~
<ins>+      if len(aBuf) > 1:</ins>
<ins>+             if (aBuf[0] == 0x202) and \</ins>
<ins>+                (aBuf[1] >= 0x9F) and \</ins>
<ins>+                (aBuf[1] <= 0xF1):</ins>
<ins>+                return aBuf[1] - 0x9F, charLen</ins>

          return -1, charLen

  class EUCJPContextAnalysis(JapaneseContextAnalysis):
~~-     def get_order(self, aStr):~~
~~-      if not aStr: return -1, 1~~
<ins>+     def get_order(self, aBuf):</ins>
<ins>+      if not aBuf: return -1, 1</ins>
          # find out current char's byte length
~~-         if (aStr[0] == '\x8E') or \~~
~~-           ((aStr[0] >= '\xA1') and (aStr[0] <= '\xFE')):~~
<ins>+         if (aBuf[0] == 0x8E) or \</ins>
<ins>+           ((aBuf[0] >= 0xA1) and (aBuf[0] <= 0xFE)):</ins>
              charLen = 2
~~-         elif aStr[0] == '\x8F':~~
<ins>+         elif aBuf[0] == 0x8F:</ins>
              charLen = 3
          else:
              charLen = 1

        # return its order if it is hiragana
~~-    if len(aStr) > 1:~~
~~-           if (aStr[0] == '\xA4') and \~~
~~-              (aStr[1] >= '\xA1') and \~~
~~-              (aStr[1] <= '\xF3'):~~
~~-                 return ord(aStr[1]) - 0xA1, charLen~~
<ins>+    if len(aBuf) > 1:</ins>
<ins>+           if (aBuf[0] == 0xA4) and \</ins>
<ins>+              (aBuf[1] >= 0xA1) and \</ins>
<ins>+              (aBuf[1] <= 0xF3):</ins>
<ins>+               return aBuf[1] - 0xA1, charLen</ins>

        return -1, charLen
```

在代码库中查找 `ord()` 函数，我们在 `chardistribution.py` 中也发现了同样的问题（更确切地说，在以下这些类中，`EUCTWDistributionAnalysis`，`EUCKRDistributionAnalysis`，`GB2312DistributionAnalysis`，`Big5DistributionAnalysis`，`SJISDistributionAnalysis` 和 `EUCJPDistributionAnalysis`）。对于它们存在的问题，解决办法与我们对 `jpcntx.py` 中的类 `EUCJPContextAnalysis` 和 `SJISContextAnalysis` 的做法相似。

Global name `'reduce'` is not defined

再次陷入中断…

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    u.close()
  File "C:\home\chardet\chardet\universaldetector.py", line 141, in close
    proberConfidence = prober.get_confidence()
  File "C:\home\chardet\chardet\latin1prober.py", line 126, in get_confidence
    total = reduce(operator.add, self._mFreqCounter)
NameError: global name 'reduce' is not defined
```

根据官方手册：What’s New In Python 3.0，函数 `reduce()` 已经从全局名字空间中移出，放到了 `functools` 模块中。引用手册中的内容：“如果需要，请使用 `functools.reduce()`，99% 的情况下，显式的 `for` 循环使代码更有可读性。”你可以从Guido van Rossum的一篇日志中看到关于这项决策的更多细节：The fate of reduce() in Python 3000。

```
def get_confidence(self):
    if self.get_state() == constants.eNotMe:
        return 0.01

 <mark>total = reduce(operator.add, self._mFreqCounter)</mark>
```

`reduce()` 函数使用两个参数 — 一个函数，一个列表（更严格地说，可迭代的对象就行了）— 然后将函数增量式地作用在列表的每个元素上。换句话说，这是一种良好而高效的用于综合(add up)列表所有元素并返回其结果的方法。

这种强大的技术使用如此频繁，所以Python就添加了一个全局的 `sum()` 函数。

```
 def get_confidence(self):
      if self.get_state() == constants.eNotMe:
          return 0.01

~~-     total = reduce(operator.add, self._mFreqCounter)~~
<ins>+     total = sum(self._mFreqCounter)</ins>
```

由于我们不再使用 `operator` 模块，所以可以在文件最上方移除那条 `import` 语句。

```
 from .charsetprober import CharSetProber
  from . import constants
~~- import operator~~
```

可以开始测试了吧？（快要吐血的样子...）

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                       ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml                             Big5 with confidence 0.99
tests\Big5\blog.worren.net.xml                               Big5 with confidence 0.99
tests\Big5\carbonxiv.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\catshadow.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\coolloud.org.tw.xml                               Big5 with confidence 0.99
tests\Big5\digitalwall.com.xml                               Big5 with confidence 0.99
tests\Big5\ebao.us.xml                                       Big5 with confidence 0.99
tests\Big5\fudesign.blogspot.com.xml                         Big5 with confidence 0.99
tests\Big5\kafkatseng.blogspot.com.xml                       Big5 with confidence 0.99
tests\Big5\ke207.blogspot.com.xml                            Big5 with confidence 0.99
tests\Big5\leavesth.blogspot.com.xml                         Big5 with confidence 0.99
tests\Big5\letterlego.blogspot.com.xml                       Big5 with confidence 0.99
tests\Big5\linyijen.blogspot.com.xml                         Big5 with confidence 0.99
tests\Big5\marilynwu.blogspot.com.xml                        Big5 with confidence 0.99
tests\Big5\myblog.pchome.com.tw.xml                          Big5 with confidence 0.99
tests\Big5\oui-design.com.xml                                Big5 with confidence 0.99
tests\Big5\sanwenji.blogspot.com.xml                         Big5 with confidence 0.99
tests\Big5\sinica.edu.tw.xml                                 Big5 with confidence 0.99
tests\Big5\sylvia1976.blogspot.com.xml                       Big5 with confidence 0.99
tests\Big5\tlkkuo.blogspot.com.xml                           Big5 with confidence 0.99
tests\Big5\tw.blog.xubg.com.xml                              Big5 with confidence 0.99
tests\Big5\unoriginalblog.com.xml                            Big5 with confidence 0.99
tests\Big5\upsaid.com.xml                                    Big5 with confidence 0.99
tests\Big5\willythecop.blogspot.com.xml                      Big5 with confidence 0.99
tests\Big5\ytc.blogspot.com.xml                              Big5 with confidence 0.99
tests\EUC-JP\aivy.co.jp.xml                                  EUC-JP with confidence 0.99
tests\EUC-JP\akaname.main.jp.xml                             EUC-JP with confidence 0.99
tests\EUC-JP\arclamp.jp.xml                                  EUC-JP with confidence 0.99
.
.
.
316 tests
```

天哪，伙计，她真的欢快地跑起来了！/me does a little dance

总结

我们学到了什么？

1. 尝试大批量地把代码从Python 2移植到Python 3上是一件让人头疼的工作。没有捷径。它确实很困难。
2. 自动化的 `2to3` 脚本确实有用，但是它只能做一些简单的辅助工作 — 函数重命名，模块重命名，语法修改等。之前，它被认为是一项会让人印象深刻的大工程，但是最后，实际上它只是一个能智能地执行查找替换机器人。
3. 在移植 `chardet` 库的时候遇到的头号问题就是：字符串和字节对象之间的差异。在我们这个情况中，这种问题比较明显，因为整个 `chardet` 库就是一直在执行从字节流到字符串的转换。但是“字节流”出现的方式会远超出你的想象。以“二进制”模式读取文件？我们

会获得字节流。获取一份web页面？调用web API？这也会返回字节流。

4. 你需要彻底地了解所面对的程序。如果那段程序是自己写自然非常好，但是至少，我们需要够理解所有晦涩难懂的细节。因为bug可能埋伏在任何地方。
5. 测试样例是必要的。没有它们的话不要尝试着移植代码。我自信移植后的 `chardet` 模块能在Python 3中工作的唯一理由是，我一开始就使用了测试集合来检验所有主要的代码路径。如果你还没有任何测试集，在移植代码之前自己写一些吧。如果你的测试集合太小，那么请写全。如果测试集够了，那么，我们就又可以开始历险了。

Chapter 16 打包 Python 类库

> " You'll find the shame is like the pain; you only feel it once. " — Marquise de Merteuil, Dangerous Liaisons

深入

读到这里，你可能是想要发布一个 Python 脚本，库，框架，或者应用程序。太棒了！世界需要更多的Python代码。

Python 3 自带一个名为 Distutils 的打包框架。Distutils 包含许多功能：构建工具（为你所准备），安装工具（为用户所准备），数据包格式（为搜索引擎所准备）等。它集成了 Python 安装包索引（"PyPI"），一个开源 Python 类库的中央资料库。

这些 Distutils 的不同功能以*setup script*为中心，一般被命名为 `setup.py` 。事实上，你已经在本书中见过一些 Distutils 安装脚本。在《HTTP Web Services》一章中，我们使用 Distutils 来安装 `httplib2` ，而在《案例研究：将 `chardet` 移植到 Python 3》一章中，我们用它安装 `chardet` 。

在本章中，你将学习 `chardet` 和 `httplib2` 的安装脚本如何工作，并将逐步（学会）发布自己的 Python 软件。

```
# chardet's setup.py
from distutils.core import setup
setup(
    name = "chardet",
    packages = ["chardet"],
    version = "1.0.2",
    description = "Universal encoding detector",
    author = "Mark Pilgrim",
    author_email = "mark@diveintomark.org",
    url = "http://chardet.feedparser.org/",
    download_url = "http://chardet.feedparser.org/download/python3-chardet-1.0.1.tgz",
    keywords = ["encoding", "i18n", "xml"],
    classifiers = [
        "Programming Language :: Python",
        "Programming Language :: Python :: 3",
        "Development Status :: 4 - Beta",
        "Environment :: Other Environment",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: GNU Library or Lesser General Public License (LGPL)",
        "Operating System :: OS Independent",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Text Processing :: Linguistic",
        ],
    long_description = """\
Universal character encoding detector
-------------------------------------

Detects
 - ASCII, UTF-8, UTF-16 (2 variants), UTF-32 (4 variants)
 - Big5, GB2312, EUC-TW, HZ-GB-2312, ISO-2022-CN (Traditional and Simplified Chinese)
 - EUC-JP, SHIFT_JIS, ISO-2022-JP (Japanese)
 - EUC-KR, ISO-2022-KR (Korean)
 - KOI8-R, MacCyrillic, IBM855, IBM866, ISO-8859-5, windows-1251 (Cyrillic)
 - ISO-8859-2, windows-1250 (Hungarian)
 - ISO-8859-5, windows-1251 (Bulgarian)
 - windows-1252 (English)
 - ISO-8859-7, windows-1253 (Greek)
 - ISO-8859-8, windows-1255 (Visual and Logical Hebrew)
 - TIS-620 (Thai)

This version requires Python 3 or later; a Python 2 version is available separately.
"""
)
```

> ☞ `chardet` 和 `httplib2` 都是开源的，但这并没有要求你在特定的许可下发布你自己的 Python 库。本章所描述的过程对任何 Python软件都适用，无论它使用什么许可证

Distutils 无法为你完成的工作

发布第一个 Python 包是一项艰巨的过程。（发布第二个相对容易一些。）Distutils 试图尽可能多的自动完成一些工作，但是仍然有一些事情你必须自己做。

- 选择一种许可协议 。. 这是一个复杂的话题，充满了派别斗争和危险。如果想将软件发布为开源软件，我冒昧地提出五点忠告：
  1. 不要撰写自己的许可证。
  2. 不要撰写自己的许可证。
  3. 不要撰写自己的许可证。
  4. 许可证并不一定必须是 GPL，但它需要与 GPL 兼容 。

5. 不要撰写自己的许可证。

- 使用 PyPI 分类系统对软件进行分类。我将在本章后面的部分解释这是什么意思。
- 写“自述”**(read me)**文件 。不要在这一点吝惜精力投入。至少，它应该让你的用户了解你的软件可以干什么并知道如何安装它。

目录结构

要开始打包 Python 软件，必须先将文件和目录安排好。 `httplib2` 的目录树如下：

```
|
+--__init__.py
|
+--iri2uri.py
```

1. 创建根目录来保存所有的目录和文件。将其以 Python 模块的名字命名。
2. 为了适应 Windows 用户，"自述"文件应包含 `.txt` 扩展名，而且它应该使用 Windows 风格回车符。不能仅仅因为你使用了一个优秀的文本编辑器，它从命令行运行并包括它自己的宏语言，而需要让你的用户为难。（你的用户使用记事本。虽然可悲，但却是事实。）即使你工作在 Linux 或 Mac OS X 环境下，优秀的文本编辑器毫无疑问地会有一个选项，允许将文件以 Windows 风格回车符来保存。
3. Distutils 安装脚本应命名为 setup.py，除非你有一个很好的理由不这样做。但你并没有一个很好的理由不这样做。
4. 如果你的Python软件只包含一个单一的 `.py` 文件，你应该把它和"自述"文件以及安装脚本放到根目录下。但 `httplib2` 并不是单一的 `.py` 文件，它是一个多文件模块 。但是没关系！只需在根目录下放置 `httplib2` 目录，这样在 `httplib2/` 根目录下就会有一个包含 `__init__.py` 文件的 `httplib2/` 目录。这并不是一个难题，事实上，它可以简化打包过程。

`chardet` 目录看起来有些不同。像 `httplib2` 一样，它是一个多文件模块 ，所以在 `chardet/` 根目录下有一个 `chardet/` 目录。除了 `README.txt` 文件，在 `docs/` 目录下，`chardet` 还有 HTML ——格式化文档。该 `docs/` 目录包含多个 `.html` 和 `.css` 文件和 `images/` 子目录，其中包含几个 `.png` 和 `.gif` 文件。（稍后你会发现，这将是很重要的。）此外，对于 (L)GPL 许可的软件，它包含一个单独的 `COPYING.txt` 文件，其中包含 LGPL 许可证的完整内容。

```
chardet/
|
+--COPYING.txt
|
+--setup.py
|
+--README.txt
|
+--docs/
|  |
|  +--index.html
|  |
|  +--usage.html
|  |
|  +--images/ ...
|
+--chardet/
   |
   +--__init__.py
   |
   +--big5freq.py
   |
   +--...
```

编写安装脚本

Distutils 安装脚本是一份 Python 脚本。从理论上讲，它可以做任何 Python 可以做的事情。在实践中，安装脚本应该做尽可能少的事情并尽可能按标准的方式做。安装脚本应该简单。安装过程越奇异，错误报告也会更奇特。

每个 Distutils 安装脚本的第一行总是相同的：

```
from distutils.core import setup
```

该行导入 `setup()` 函数，这是 Distutils 的主入口点。95% 的 Distutils 安装脚本仅由一个对 `setup()` 方法的调用组成。（这完全是我臆造的统计，但如果你的 Distutils 安装脚本所做的比仅仅调用 `setup()` 方法更多，你会有一个好的理由。你有一个好的理由吗？我并不这么认为。）

`setup()` 方法可以有几十个参数 。为了使每个参与者都能清楚，你必须对每个参数使用命名变量 。这不只是一项约定，还是一项硬性要求。如果尝试以非命名变量调用 `setup()` 方法，安装脚本会崩溃。

下面的命名变量是必需的：

- **name**，安装包的名称。
- **version**，安装包的版本。
- **author**，您的全名。
- **author_email**，您的邮件地址。
- **url**，项目主页。如果没有一个单独的项目网站，这里可以是安装包的 PyPI 的页面地址。

虽然以下内容不是必须的，但我也建议你把他们包括在你的安装脚本里：

- **description**，在线的项目摘要。
- **long_description**，以 reStructuredText format 格式编写的多行字符串。PyPI 将其转换为 HTML 并在安装包中显示它。
- **classifiers**，下一节中将讲述的特别格式化字符串。

> ☞安装脚本中用到的元数据具体定义在 PEP 314 中。

现在让我们看看 `chardet` 的安装脚本。它包含所有这些要求的和建议的参数，还有一个我没有提到： `packages` 。

```
from distutils.core import setup
setup(
    name = 'chardet',
    <mark>packages = ['chardet']</mark>,
    version = '1.0.2',
    description = 'Universal encoding detector',
    author='Mark Pilgrim',
    ...
)
```

在分发过程中，这个 `packages` 参数凸显出一个不幸的词汇表重叠。我们一直在谈论正在构建的“安装包”（并将潜在地出现在Python包索引中）。但是，这并不是 `packages` 参数所指代的。它指代的是 `chardet` 模块是一个多文件模块这一事实，有时也被称为…“包”。 `packages` 参数告诉 Distutils 去包含 `chardet/` 目录，它的 `__init__.py` 文件，以及所有其他构成 `chardet` 模块的 `.py` 文件。这还算比较重要；如果你忘记了包含实际的代码，那么所有这些关于文件和元数据的愉快交谈都将是无关紧要的。

将包分类

Python 包索引（“PyPI”）包含成千上万的 Python 库。正确的分类数据将让人们更容易找到你的包。PyPI 让你以类别的形式浏览包 。你甚至可以选择多个类别来缩小搜索范围。分类不是你可以忽略的不可见的元数据！

你可以通过传递 `classifiers` 参数给 Distutils 的 `setup()` 方法来给你的软件分类。 `classifers` 参数是一个字符串列表。这些字符串不是任意形式的。所有的分类字符串应该来自 PyPI 上的列表 。

分类是可选的。你可以写一个不包含任何分类的 Distutils 安装脚本。不要这样做。 你应该总是至少包括以下分类：

- <b0编程语言. 特别的，你应该包括 `"Programming Language :: Python"` 和" `Programming Language :: Python :: 3` "。如果你不包括这些，你的包将不会出现在兼容Python 3的库列表中，它链接自每个 `pypi.python.org` 单页的侧边栏。

- 许可证. 当我评价一个第三方库的时候，这绝对是我寻找的第一个东西。不要让我（花太多时间）寻找这个重要的信息。不要包含一个以上的许可证分类，除非你的软件明确地在多许可证下分发。（不要在多许可证下发布你的软件，除非你不得不这样做。不要强迫别人这样做。许可证已经足够让人头痛了，不要使情况变得更糟。）
- 操作系统**0**. 如果你的软件只能运行于Windows（或Mac OS X或Linux），我想要尽早知道。如果你的软件不包含任何特定平台的代码并可以在任何平台运行，请使用分类 `"Operating System :: OS Independent"` 。 `多操作系统` 分类仅在你的软件在不同平台需要特别支持时使用。（这并不常见。）

我还建议你包括以下分类：

- 开发状态. 你的软件品质适合beta发布么？适合Alpha发布么？还是Pre-alpha？在这里面选择一个吧。要诚实点。
- 目标用户. 谁会下载你的软件？最常见的选项包括： `Developers` 、 `End Users/Desktop` 、 `Science/Research` 和 `System Administrators` 。
- 框架. 如果你的软件是像 Django 或 Zope 这样较大的框架的插件，请包含适当的 `Framework` 分类。如果不是，请忽略它。
- 主题. 有 大量的主题 可供选择，选择所有的适用项。

包分类的优秀范例

作为例子，下面是 Django 的分类。它是一个运行在 Web 服务器上的，可用于生产环境的，跨平台的，使用 BSD 授权的 Web 应用程序框架。（Django还没有与Python 3兼容，因此，并没有列出 `Programming Language :: Python :: 3` 分类。）

```
Programming Language :: Python
License :: OSI Approved :: BSD License
Operating System :: OS Independent
Development Status :: 5 - Production/Stable
Environment :: Web Environment
Framework :: Django
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Internet :: WWW/HTTP :: Dynamic Content
Topic :: Internet :: WWW/HTTP :: WSGI
Topic :: Software Development :: Libraries :: Python Modules
```

下面是 `chardet` 的分类。它就是在《案例研究：将 `chardet` 移植到 Python 3》一章提到的字符编码检测库。 `chardet` 是高质量的，跨平台的，与 Python 3 兼容的， LGPL 许可的库。它旨在让开发者将其集成进自己的产品。

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: GNU Library or Lesser General Public License (LGPL)
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Other Environment
Intended Audience :: Developers
Topic :: Text Processing :: Linguistic
Topic :: Software Development :: Libraries :: Python Modules
```

以下是在本章开头我提到的 `httplib2` 模块——HTTP 的分类。 `httplib2` 是一个测试品质的，跨平台的，MIT 许可证授权的，为 Python 开发者准备的模块。

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: MIT License
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Web Environment
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Software Development :: Libraries :: Python Modules
```

通过清单指定附加文件

默认情况下，Distutils 将把下列文件包含在你的发布包中：

- `README.txt`
- `setup.py`
- 由列在 `packages` 参数中的多模块文件所需的 `.py` 文件
- 在 `py_modules` 参数中列出的单独 `.py` 文件

这将覆盖 `httplib2` 项目的所有文件。但对于 `chardet` 项目，我们还希望包含 `COPYING.txt` 许可文件和含有图像与 HTML 文件的整个 `docs/` 目录。要让 Distutils 在构建 `chardet` 发布包时包含这些额外的文件和目录，你需要创建一个 *manifest file* 。

清单文件是一个名为 `MANIFEST.in` 的文本文件。将它放置在项目的根目录下，同 `README.txt` 和 `setup.py` 一起。清单文件并不是 Python 脚本，它是文本文件，其中包含一系列 Distutils 定义格式的命令。清单命令允许你包含或排除特定的文件和目录。

以下是 `chardet` 项目的全部清单文件：

1. 第一行是不言自明的：包含项目根目录的 `COPYING.txt` 文件。
2. 第二行有些复杂。 `recursive-include` 命令需要一个目录名和至少一个文件名。文件名并不限于特定的文件，可以包含通配符。这行的意思是“看到在项目根目录下的 `docs/` 目录了吗？在该目录下（递归地）查找 `.html` 、 `.css` 、 `.png` 和 `.gif` 文件。我希望将他们都包含在我的发布包中。”

所有的清单命令都将保持你在项目目录中所设置的目录结构。 `recursive-include` 命令不会将一组 `.html` 和 `.png` 文件放置在你的发布包的根目录下。它将保持现有的 `docs/` 目录结构，但只包含该目录内匹配给定的通配符的文件。（之前我并没有提到， `chardet` 的文档实际上由 XML 语言写成，并由一个单独的脚本转换为 HTML 。我不想在发布包中包含XML 文件，只包含 HTML 文件和图像。）

> ☞清单文件有自己独特的格式。详见 分发指定文件 和清单文件命令。

重申：仅仅在你需要包含一些 Distutils 不会默认包含的文件时才创建清单文件。I如果你确实需要一个清单文件，它应该只包含那些Distutils不会自动包含的文件和目录。

检查安装脚本的错误

有许多事情需要留意。Distutils带有一个内置的验证命令，它检查是否所有必须的元数据都体现在你的安装脚本中。例如，如果你忘记包含 `version` 参数，Distutils 会提醒你。

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check
running check
warning: check: missing required meta-data: version
```

当你包含了 `version` 参数（和所有其他所需的元数据）时， `check` 命令将如下所示：

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check
running check
```

创建发布源

Distutils 支持构建多种类型的发布包。至少，你应该建立一个“源代码分发”，其中包含源代码，你的Distutils 安装脚本，“read me ”文件和你想要包含其他文件 。为了建立一个源代码分发，传递 `sdist` 命令给你的 Distutils 安装脚本。

```
c:\Users\pilgrim\chardet> <mark>c:\python31\python.exe setup.py sdist</mark>
running sdist
running check
reading manifest template 'MANIFEST.in'
writing manifest file 'MANIFEST'
creating chardet-1.0.2
creating chardet-1.0.2\chardet
creating chardet-1.0.2\docs
creating chardet-1.0.2\docs\images
copying files to chardet-1.0.2...
copying COPYING -> chardet-1.0.2
copying README.txt -> chardet-1.0.2
copying setup.py -> chardet-1.0.2
copying chardet\__init__.py -> chardet-1.0.2\chardet
copying chardet\big5freq.py -> chardet-1.0.2\chardet
...
copying chardet\universaldetector.py -> chardet-1.0.2\chardet
copying chardet\utf8prober.py -> chardet-1.0.2\chardet
copying docs\faq.html -> chardet-1.0.2\docs
copying docs\history.html -> chardet-1.0.2\docs
copying docs\how-it-works.html -> chardet-1.0.2\docs
copying docs\index.html -> chardet-1.0.2\docs
copying docs\license.html -> chardet-1.0.2\docs
copying docs\supported-encodings.html -> chardet-1.0.2\docs
copying docs\usage.html -> chardet-1.0.2\docs
copying docs\images\caution.png -> chardet-1.0.2\docs\images
copying docs\images\important.png -> chardet-1.0.2\docs\images
copying docs\images\note.png -> chardet-1.0.2\docs\images
copying docs\images\permalink.gif -> chardet-1.0.2\docs\images
copying docs\images\tip.png -> chardet-1.0.2\docs\images
copying docs\images\warning.png -> chardet-1.0.2\docs\images
creating dist
creating 'dist\chardet-1.0.2.zip' and adding 'chardet-1.0.2' to it
adding 'chardet-1.0.2\COPYING'
adding 'chardet-1.0.2\PKG-INFO'
adding 'chardet-1.0.2\README.txt'
adding 'chardet-1.0.2\setup.py'
adding 'chardet-1.0.2\chardet\big5freq.py'
adding 'chardet-1.0.2\chardet\big5prober.py'
...
adding 'chardet-1.0.2\chardet\universaldetector.py'
adding 'chardet-1.0.2\chardet\utf8prober.py'
adding 'chardet-1.0.2\chardet\__init__.py'
adding 'chardet-1.0.2\docs\faq.html'
adding 'chardet-1.0.2\docs\history.html'
adding 'chardet-1.0.2\docs\how-it-works.html'
adding 'chardet-1.0.2\docs\index.html'
adding 'chardet-1.0.2\docs\license.html'
adding 'chardet-1.0.2\docs\supported-encodings.html'
adding 'chardet-1.0.2\docs\usage.html'
adding 'chardet-1.0.2\docs\images\caution.png'
adding 'chardet-1.0.2\docs\images\important.png'
adding 'chardet-1.0.2\docs\images\note.png'
adding 'chardet-1.0.2\docs\images\permalink.gif'
adding 'chardet-1.0.2\docs\images\tip.png'
adding 'chardet-1.0.2\docs\images\warning.png'
removing 'chardet-1.0.2' (and everything under it)
```

有几件事情需要注意：

- Distutils发现了清单文件(`MANIFEST.in`)
- Distutils 成功地解析了清单文件，并添加了我们所需要的文件—— `COPYING.txt` 和在 `docs/` 目录下的 HTML 与图像文件。
- 如果你进入你的项目目录，你会看到 Distutils 创建了一个 `dist/` 目录。你可以分发在 `dist/` 目录中的 `.zip` 文件。

```
c:\Users\pilgrim\chardet> <mark>dir dist</mark>
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\chardet\dist

07/30/2009  06:29 PM    <DIR>          .
07/30/2009  06:29 PM    <DIR>          ..
07/30/2009  06:29 PM           206,440 <mark>chardet-1.0.2.zip</mark>
               1 File(s)        206,440 bytes
               2 Dir(s)  61,424,635,904 bytes free
```

创建图形化安装程序

在我看来，每一个 Python 库都应该为 Windows 用户提供图形安装程序。这很容易做（即使你并没有运行 Windows ），而且 Windows 用户会对此表示感激。

通过传递 `bdist_wininst` 命令到你的 Distutils 安装脚本，它可以为你创建一个图形化的 Windows 安装程序 。

```
c:\Users\pilgrim\chardet> <mark>c:\python31\python.exe setup.py bdist_wininst</mark>
running bdist_wininst
running build
running build_py
creating build
creating build\lib
creating build\lib\chardet
copying chardet\big5freq.py -> build\lib\chardet
copying chardet\big5prober.py -> build\lib\chardet
...
copying chardet\universaldetector.py -> build\lib\chardet
copying chardet\utf8prober.py -> build\lib\chardet
copying chardet\__init__.py -> build\lib\chardet
installing to build\bdist.win32\wininst
running install_lib
creating build\bdist.win32
creating build\bdist.win32\wininst
creating build\bdist.win32\wininst\PURELIB
creating build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\big5freq.py -> build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\big5prober.py -> build\bdist.win32\wininst\PURELIB\chardet
...
copying build\lib\chardet\universaldetector.py -> build\bdist.win32\wininst\PURELIB\chard
copying build\lib\chardet\utf8prober.py -> build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\__init__.py -> build\bdist.win32\wininst\PURELIB\chardet
running install_egg_info
Writing build\bdist.win32\wininst\PURELIB\chardet-1.0.2-py3.1.egg-info
creating 'c:\users\pilgrim\appdata\local\temp\tmp2f4h7e.zip' and adding '.' to it
adding 'PURELIB\chardet-1.0.2-py3.1.egg-info'
adding 'PURELIB\chardet\big5freq.py'
adding 'PURELIB\chardet\big5prober.py'
...
adding 'PURELIB\chardet\universaldetector.py'
adding 'PURELIB\chardet\utf8prober.py'
adding 'PURELIB\chardet\__init__.py'
removing 'build\bdist.win32\wininst' (and everything under it)
c:\Users\pilgrim\chardet> <mark>dir dist</mark>
c:\Users\pilgrim\chardet>dir dist
 Volume in drive C has no label.
 Volume Serial Number is AADE-E29F

 Directory of c:\Users\pilgrim\chardet\dist

07/30/2009  10:14 PM    <DIR>          .
07/30/2009  10:14 PM    <DIR>          ..
07/30/2009  10:14 PM           371,236 <mark>chardet-1.0.2.win32.exe</mark>
07/30/2009  06:29 PM           206,440 chardet-1.0.2.zip
               2 File(s)        577,676 bytes
               2 Dir(s)  61,424,070,656 bytes free
```

为其它操作系统编译安装包

Distutils 可以帮助你为 Linux 用户构建可安装包 。我认为，这可能不值得你浪费时间。如果你希望在 Linux 中分发你的软件，你最好将时间花在与那些社区成员进行交流上，他们专门为主流 Linux 发行版打包软件。

例如，我的 `chardet` 库包含在 Debian GNU/Linux 软件仓库中（因而也包含在 Ubuntu 的软件仓库中）。我不曾做任何事情，我只在那里将安装包展示了一天。Debian 社区拥有他们自己的关于打包 Python 库的政策，并且Debian 的 `python-chardet` 包被设计为遵循这些公约。由于这个包存在在 Debian 的软件仓库中，依赖于 Debian 用户所选择的管理自己计算机的系统设置，他们会收到该包的安全更新和（或）新版本。

Distutils构建的包不具有Linux包所提供的任何优势。你的时间最好花在其他地方。

将软件添加到 Python 安装包列表

上传软件到 Python 包索引需要三个步骤。

1. 注册你自己
2. 注册你的软件
3. 上传你通过 `setup.py sdist` 和 `setup.py bdist_*` 创建的包。

要注册自己，访问 PyPI用户注册页面。输入你想要的用户名和密码，提供一个有效的电子邮件地址，然后点击 `Register` 按钮。（如果你有一个 PGP 或 GPG 密钥，你也可以提供。如果你没有或者不知道这是什么意思，不用担心。）检查你的电子邮件，在几分钟之内，你应该会收到一封来自 PyPI 的包含验证链接的邮件。点击链接以完成注册过程。

现在，你需要在PyPI注册你的软件并上传它。你可以用一步完成。

```
running register
We need to know who you are, so please choose either:
 1\. use your existing login,
 2\. register as a new user,
 3\. have the server generate a new password for you (and email it to you), or
 4\. quit

Password:

Server response (200): OK

... output trimmed for brevity ...

... output trimmed for brevity ...

Submitting dist\chardet-1.0.2.zip to http://pypi.python.org/pypi
Server response (200): OK
Submitting dist\chardet-1.0.2.win32.exe to http://pypi.python.org/pypi
Server response (200): OK
I can store your PyPI login so future submissions will be faster.
(the login will be stored in c:\home\.pypirc)
```

1. 当你第一次发布你的项目时，Distutils 会将你的软件加入到Python包索引中并给出它的URL。在这之后，它只会用你在 `setup.py` 参数所做的任何改变来更新项目的元数据。之后，它构建一个源代码发布 (`sdist`) 和一个 Windows 安装程序 (`bdist_wininst`) 并把他们上传到PyPI (`upload`)。
2. 键入 `1` 或 `ENTER` 选择“ 使用已有的账户登录【use your existing login.】”。
3. 输入你在 PyPI 用户注册页面所选择的用户名和密码。Distuils不会回显你的密码，它甚至不会在相应的位置显示星号。只需输入你的密码，然后按 `回车键` 。
4. Distutils 在 Python 包索引注册你的包……
5. ……构建源代码分发……
6. ……构建Windows安装程序……

7. ……并把它们上传至 Python 包索引。
8. 如果你想自动完成发布新版本的过程，你需要将你的PyPI凭据保存在一个本地文件中。这完全是不安全的而且是完全可选的。

恭喜你，现在，在Python包索引中有你自己的页面了！地址是 `http://pypi.python.org/pypi/_NAME_`，其中 *NAME* 是你在 `setup.py` 文件中 `name` 参数所传递的字符串。

如果你想发布一个新版本，只需以新的版本号更新 `setup.py` 文件，然后再一次运行相同的上传命令:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist_wininst up
```

Python 打包工具的一些可能的将来

Distutils并非是一个代替所有并终结所有的 Python 打包，但在写本书时（2009年8月），它是唯一可以工作在 Python 3 下的打包框架。对于Python 2，还有许多其他的框架，有的重在安装，有的重在测试，还有的重在部署。在未来，它们中的一部分或全体都将移植到Python 3。

以下框架重在安装：

- Setuptools
- Pip
- Distribute

以下框架重在测试和部署：

- `virtualenv`
- `zc.buildout`
- Paver
- Fabric
- `py2exe`

深入阅读

关于 Distutils：

- 通过 Distutils 发布 Python 模块
- 核心发布功能 列出了 `setup()` 函数的所有可能参数
- Distutils 食谱
- PEP 370: 每用户 `site-packages` 目录

- PEP 370 和 “environment stew”

其它打包框架：

- Python 打包生态系统
- 关于打包
- 对 “关于打包” 的几点纠错
- 我为什么喜欢 Pip
- Python 打包：几点看法
- 没有人期望 Python 打包！

你的位置: Home ‣ Dive Into Python 3 ‣

难度等级: ♦♦♦♦♦

Chapter A 使用 `2to3` 将代码移植到Python 3

> " Life is pleasant. Death is peaceful. It's the transition that's troublesome. " — Isaac Asimov (attributed)

概述

几乎所有的Python 2程序都需要一些修改才能正常地运行在Python 3的环境下。为了简化这个转换过程，Python 3自带了一个叫做 `2to3` 的实用脚本(Utility Script)，这个脚本会将你的Python 2程序源文件作为输入，然后自动将其转换到Python 3的形式。案例研究: 将 `chardet` 移植到Python 3(porting chardet to Python 3)描述了如何运行这个脚本，然后展示了一些它不能自动修复的情况。这篇附录描述了它能够自动修复的内容。

`print` 语句

在Python 2里， `print` 是一个语句。无论你想输出什么，只要将它们放在 `print` 关键字后边就可以。在Python 3里， `print()` 是一个函数。就像其他的函数一样， `print()` 需要你将想要输出的东西作为参数传给它。

Notes	Python 2	Python 3
①	`print`	`print()`
②	`print 1`	`print(1)`
③	`print 1, 2`	`print(1, 2)`
④	`print 1, 2,`	`print(1, 2, end=' ')`
⑤	`print >>sys.stderr, 1, 2, 3`	`print(1, 2, 3, file=sys.stderr)`

1. 为输出一个空白行，需要调用不带参数的 `print()` 。
2. 为输出一个单独的值，需要将这这个值作为 `print()` 的一个参数就可以了。
3. 为输出使用一个空格分隔的两个值，用两个参数调用 `print()` 即可。
4. 这个例子有一些技巧。在Python 2里，如果你使用一个逗号(,)作为 `print` 语句的结尾，它将会用空格分隔输出的结果，然后在输出一个尾随的空格(trailing space)，而不输出回车(carriage return)。在Python 3里，通过把 `end=' '` 作为一个关键字参数传给 `print()` 可以实现同样的效果。参数 `end` 的默认值为 `'\n'` ，所以通过重新指定 `end` 参数的值，可以取消在末尾输出回车符。
5. 在Python 2里，你可以通过使用 `>>pipe_name` 语法，把输出重定向到一个管道，比如 `sys.stderr` 。在Python 3里，你可以通过将管道作为关键字参数 `file` 的值传递给 `print()` 来完成同样的功能。参数 `file` 的默认值为 `std.stdout` ，所以重新指定它的

值将会使 `print()` 输出到一个另外一个管道。

Unicode字符串

Python 2有两种字符串类型：Unicode字符串和非Unicode字符串。Python 3只有一种类型：Unicode字符串(Unicode strings)。

Notes	Python 2	Python 3
①	`u'PapayaWhip'`	`'PapayaWhip'`
②	`ur'PapayaWhip\foo'`	`r'PapayaWhip\foo'`

1. Python 2里的Unicode字符串在Python 3里即普通字符串，因为在Python 3里字符串总是Unicode形式的。
2. Unicode原始字符串(raw string)(使用这种字符串，Python不会自动转义反斜线"\")也被替换为普通的字符串，因为在Python 3里，所有原始字符串都是以Unicode编码的。

全局函数 `unicode()`

Python 2有两个全局函数可以把对象强制转换成字符串：`unicode()` 把对象转换成Unicode字符串，还有 `str()` 把对象转换为非Unicode字符串。Python 3只有一种字符串类型，Unicode字符串，所以 `str()` 函数即可完成所有的功能。(`unicode()` 函数在Python 3里不再存在了。)

Notes	Python 2	Python 3
	`unicode(anything)`	`str(anything)`

`long` 长整型

Python 2有为非浮点数准备的 `int` 和 `long` 类型。 `int` 类型的最大值不能超过 `sys.maxint`，而且这个最大值是平台相关的。可以通过在数字的末尾附上一个 `L` 来定义长整型，显然，它比 `int` 类型表示的数字范围更大。在Python 3里，只有一种整数类型 `int`，大多数情况下，它很像Python 2里的长整型。由于已经不存在两种类型的整数，所以就没有必要使用特殊的语法去区别他们。

进一步阅读：PEP 237：统一长整型和整型。

Notes	Python 2	Python 3
①	`x = 1000000000000L`	`x = 1000000000000`
②	`x = 0xFFFFFFFFFFFFL`	`x = 0xFFFFFFFFFFFF`
③	`long(x)`	`int(x)`
④	`type(x) is long`	`type(x) is int`
⑤	`isinstance(x, long)`	`isinstance(x, int)`

1. 在Python 2里的十进制长整型在Python 3里被替换为十进制的普通整数。
2. 在Python 2里的十六进制长整型在Python 3里被替换为十六进制的普通整数。
3. 在Python 3里，由于长整型已经不存在了，自然原来的 `long()` 函数也没有了。为了强制转换一个变量到整型，可以使用 `int()` 函数。
4. 检查一个变量是否是整型，获得它的数据类型，并与一个 `int` 类型(不是 `long`)的作比较。
5. 你也可以使用 `isinstance()` 函数来检查数据类型；再强调一次，使用 `int` ，而不是 `long` ，来检查整数类型。

<> 比较运算符

Python 2支持 `<>` 作为 `!=` 的同义词。Python 3只支持 `!=` ，不再支持<>了。

Notes	Python 2	Python 3
①	`if x <> y:`	`if x != y:`
②	`if x <> y <> z:`	`if x != y != z:`

1. 简单地比较。
2. 相对复杂的三个值之间的比较。

字典类方法 `has_key()`

在Python 2里，字典对象的 `has_key()` 方法用来测试字典是否包含特定的键(key)。Python 3不再支持这个方法了。你需要使用 `in` 运算符。

Notes	Python 2	Python
①	`a_dictionary.has_key('PapayaWhip')`	`'PapayaWhip' in a_dict`
②	`a_dictionary.has_key(x) or a_dictionary.has_key(y)`	`x in a_dictionary or y`
③	`a_dictionary.has_key(x or y)`	`(x or y) in a_dictionar`
④	`a_dictionary.has_key(x + y)`	`(x + y) in a_dictionary`
⑤	`x + a_dictionary.has_key(y)`	`x + (y in a_dictionary)`

1. 最简单的形式。
2. 运算符 `or` 的优先级高于运算符 `in` ，所以这里不需要添加括号。
3. 另一方面，出于同样的原因 — `or` 的优先级大于 `in` ，这里需要添加括号。(注意：这里的代码与前面那行完全不同。Python会先解释 `x or y` ，得到结果 `x` (如果 `x` 在布尔上下文里的值是真)或者 `y` 。然后Python检查这个结果是不是 `a_dictionary` 的一个键。)
4. 运算符 `in` 的优先级大于运算符 `+` ，所以代码里的这种形式从技术上说不需要括号，但是 `2to3` 还是添加了。
5. 这种形式一定需要括号，因为 `in` 的优先级大于 `+` 。

返回列表的字典类方法

在Python 2里，许多字典类方法的返回值是列表。其中最常用方法的有 `keys` ， `items` 和 `values` 。在Python 3里，所有以上方法的返回值改为动态视图(dynamic view)。在一些上下文环境里，这种改变并不会产生影响。如果这些方法的返回值被立即传递给另外一个函数，并且那个函数会遍历整个序列，那么以上方法的返回值是列表或者视图并不会产生什么不同。在另外一些情况下，Python 3的这些改变干系重大。如果你期待一个能被独立寻址元素的列表，那么Python 3的这些改变将会使你的代码卡住(choke)，因为视图(view)不支持索引(indexing)。

Notes	Python 2	Python 3
①	`a_dictionary.keys()`	`list(a_dictionary.keys())`
②	`a_dictionary.items()`	`list(a_dictionary.items())`
③	`a_dictionary.iterkeys()`	`iter(a_dictionary.keys())`
④	`[i for i in a_dictionary.iterkeys()]`	`[i for i in a_dictionary.keys()]`
⑤	`min(a_dictionary.keys())`	*no change*

1. 使用 `list()` 函数将 `keys()` 的返回值转换为一个静态列表，出于安全方面的考量， `2to3` 可能会报错。这样的代码是有效的，但是对于使用视图来说，它的效率低一些。你应该检查转换后的代码，看看是否一定需要列表，也许视图也能完成同样的工作。
2. 这是另外一种视图(关于 `items()` 方法的)到列表的转换。 `2to3` 对 `values()` 方法返回值的

转换也是一样的。
3. Python 3里不再支持 `iterkeys()` 了。如果必要，使用 `iter()` 将 `keys()` 的返回值转换成为一个迭代器。
4. `2to3` 能够识别出 `iterkeys()` 方法在列表解析里被使用，然后将它转换为Python 3里的 `keys()` 方法(不需要使用额外的 `iter()` 去包装其返回值)。这样是可行的，因为视图是可迭代的。
5. `2to3` 也能识别出 `keys()` 方法的返回值被立即传给另外一个会遍历整个序列的函数，所以也就没有必要先把 `keys()` 的返回值转换到一个列表。相反的，`min()` 函数会很乐意遍历视图。这个过程对 `min()`，`max()`，`sum()`，`list()`，`tuple()`，`set()`，`sorted()`，`any()` 和 `all()` 同样有效。

被重命名或者重新组织的模块

从Python 2到Python 3，标准库里的一些模块已经被重命名了。还有一些相互关联的模块也被组合或者重新组织，以使得这种关联更有逻辑性。

`http`

在Python 3里，几个相关的HTTP模块被组合成一个单独的包，即 `http` 。

Notes	Python 2	Python 3
①	`import httplib`	`import http.client`
②	`import Cookie`	`import http.cookies`
③	`import cookielib`	`import http.cookiejar`
④	`import BaseHTTPServer` `import SimpleHTTPServer` `import CGIHttpServer`	`import http.server`

1. `http.client` 模块实现了一个底层的库，可以用来请求HTTP资源，解析HTTP响应。
2. `http.cookies` 模块提供一个蟒样的(Pythonic)接口来获取通过HTTP头部(HTTP header)Set-Cookie发送的cookies
3. 常用的流行的浏览器会把cookies以文件形式存放在磁盘上，`http.cookiejar` 模块可以操作这些文件。
4. `http.server` 模块实现了一个基本的HTTP服务器

`urllib`

Python 2有一些用来分析，编码和获取URL的模块，但是这些模块就像老鼠窝一样相互重叠。在Python 3里，这些模块被重构、组合成了一个单独的包，即 `urllib` 。

Notes	Python 2	Python 3
①	`import urllib`	`import urllib.request, urllib.parse, url`
②	`import urllib2`	`import urllib.request, urllib.error`
③	`import urlparse`	`import urllib.parse`
④	`import robotparser`	`import urllib.robotparser`
⑤	`from urllib import FancyURLopener` `from urllib import urlencode`	`from urllib.request import FancyURLopene` `from urllib.parse import urlencode`
⑥	`from urllib2 import Request` `from urllib2 import HTTPError`	`from urllib.request import Request` `from urllib.error import HTTPError`

1. 以前，Python 2里的 `urllib` 模块有各种各样的函数，包括用来获取数据的 `urlopen()` ，还有用来将URL分割成其组成部分的 `splittype()` ， `splithost()` 和 `splituser()` 函数。在新的 `urllib` 包里，这些函数被组织得更有逻辑性。2to3将会修改这些函数的调用以适应新的命名方案。
2. 在Python 3里，以前的 `urllib2` 模块被并入了 `urllib` 包。同时，以 `urllib2` 里各种你最喜爱的东西将会一个不缺地出现在Python 3的 `urllib` 模块里，比如 `build_opener()` 方法， `Request` 对象， `HTTPBasicAuthHandler` 和friends。
3. Python 3里的 `urllib.parse` 模块包含了原来Python 2里 `urlparse` 模块所有的解析函数。
4. `urllib.robotparse` 模块解析 `robots.txt` 文件。
5. 处理HTTP重定向和其他状态码的 `FancyURLopener` 类在Python 3里的 `urllib.request` 模块里依然有效。 `urlencode()` 函数已经被转移到了 `urllib.parse` 里。
6. `Request` 对象在 `urllib.request` 里依然有效，但是像 `HTTPError` 这样的常量已经被转移到了 `urllib.error` 里。

我是否有提到 `2to3` 也会重写你的函数调用？比如，如果你的Python 2代码里导入了 `urllib` 模块，调用了 `urllib.urlopen()` 函数获取数据， `2to3` 会同时修改 `import` 语句和函数调用。

Notes	Python 2	
	`import urllib` `print urllib.urlopen('http://diveintopython3.org/').read()`	`import urllib.r`

dbm

所有的DBM克隆(DBM clone)现在在单独的一个包里，即 `dbm` 。如果你需要其中某个特定的变体，比如GNU DBM，你可以导入 `dbm` 包中合适的模块。

Notes	Python 2	Python 3
	`import dbm`	`import dbm.ndbm`
	`import gdbm`	`import dbm.gnu`
	`import dbhash`	`import dbm.bsd`
	`import dumbdbm`	`import dbm.dumb`
	`import anydbm` `import whichdb`	`import dbm`

`xmlrpc`

XML-RPC是一个通过HTTP协议执行远程RPC调用的轻重级方法。一些XML-RPC客户端和XML-RPC服务端的实现库现在被组合到了独立的包，即 `xmlrpc` 。

Notes	Python 2	Python 3
	`import xmlrpclib`	`import xmlrpc.client`
	`import DocXMLRPCServer` `import SimpleXMLRPCServer`	`import xmlrpc.server`

其他模块

Notes	Python 2	Python 3
①	[1]	`import io`
②	[2]	`import pickle`
③	`import __builtin__`	`import builtins`
④	`import copy_reg`	`import copyreg`
⑤	`import Queue`	`import queue`
⑥	`import SocketServer`	`import socketserver`
⑦	`import ConfigParser`	`import configparser`
⑧	`import repr`	`import reprlib`
⑨	`import commands`	`import subprocess`

[1]:

```
try:
    import cStringIO as StringIO
except ImportError:
    import StringIO
```

[2]:

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

1. 在Python 2里，你通常会这样做，首先尝试把 `cStringIO` 导入作为 `StringIO` 的替代，如果失败了，再导入 `StringIO` 。不要在Python 3里这样做；`io` 模块会帮你处理好这件事情。它会找出可用的最快实现方法，然后自动使用它。
2. 在Python 2里，导入最快的 `pickle` 实现也是一个与上边相似的能用方法。在Python 3里，`pickle` 模块会自动为你处理，所以不要再这样做。
3. `builtins` 模块包含了在整个Python语言里都会使用的全局函数，类和常量。重新定义 `builtins` 模块里的某个函数意味着在每处都重定义了这个全局函数。这听起来很强大，但是同时也是很可怕的。
4. `copyreg` 模块为用C语言定义的用户自定义类型添加了 `pickle` 模块的支持。
5. `queue` 模块实现一个生产者消费者队列(multi-producer, multi-consumer queue)。
6. `socketserver` 模块为实现各种socket server提供了通用基础类。
7. `configparser` 模块用来解析INI-style配置文件。
8. `reprlib` 模块重新实现了内置函数 `repr()` ，并添加了对字符串表示被截断前长度的控制。
9. `subprocess` 模块允许你创建子进程，连接到他们的管道，然后获取他们的返回值。

包内的相对导入

包是由一组相关联的模块共同组成的单个实体。在Python 2的时候，为了实现同一个包内模块的相互引用，你会使用 `import foo` 或者 `from foo import Bar` 。Python 2解释器会先在当前目录里搜索 `foo.py` ，然后再去Python搜索路径(`sys.path`)里搜索。在Python 3里这个过程有一点不同。Python 3不会首先在当前路径搜索，它会直接在Python的搜索路径里寻找。如果你想要包里的一个模块导入包里的另外一个模块，你需要显式地提供两个模块的相对路径。

假设你有如下包，多个文件在同一个目录下：

```
chardet/
|
+--__init__.py
|
+--constants.py
|
+--mbcharsetprober.py
|
+--universaldetector.py
```

现在假设 `universaldetector.py` 需要整个导入 `constants.py` ，另外还需要导入 `mbcharsetprober.py` 的一个类。你会怎样做?

Notes	Python 2	P
①	`import constants`	`from . import constants`
②	`from mbcharsetprober import MultiByteCharSetProber`	`from .mbcharsetprober i`

1. 当你需要从包的其他地方导入整个模块，使用新的 `from . import` 语法。这里的句号(.)即表示当前文件(`universaldetector.py`)和你想要导入文件(`constants.py`)之间的相对路径。在这个样例中，这两个文件在同一个目录里，所以使用了单个句号。你也可以从父目录(`from .. import anothermodule`)或者子目录里导入。
2. 为了将一个特定的类或者函数从其他模块里直接导入到你的模块的名字空间里，在需要导入的模块名前加上相对路径，并且去掉最后一个斜线(slash)。在这个例子中， `mbcharsetprober.py` 与 `universaldetector.py` 在同一个目录里，所以相对路径名就是一个句号。你也可以从父目录(from .. import anothermodule)或者子目录里导入。

迭代器方法 `next()`

在Python 2里，迭代器有一个 `next()` 方法，用来返回序列里的下一项。在Python 3里这同样成立，但是现在有了一个新的全局的函数 `next()` ，它使用一个迭代器作为参数。

Notes	Python 2	Python 3
①	`anIterator.next()`	`next(anIterator)`
②	`a_function_that_returns_an_iterator().next()`	`next(a_function_that_returns_`
③	[1]	[2]
④	[3]	*no change*
⑤	[4]	[5]

[1]:

```
class A:
    def next(self):
        pass
```

[2]:

```
class A:
    def __next__(self):
        pass
```

[3]:

```
class A:
    def next(self, x, y):
        pass
```

[4]:

```
next = 42
for an_iterator in a_sequence_of_iterators:
    an_iterator.next()
```

[5]:

```
next = 42
for an_iterator in a_sequence_of_iterators:
    an_iterator.__next__()
```

1. 最简单的例子，你不再调用一个迭代器的 `next()` 方法，现在你将迭代器自身作为参数传递给全局函数 `next()` 。
2. 假如你有一个返回值是迭代器的函数，调用这个函数然后把结果作为参数传递给 `next()` 函数。(`2to3` 脚本足够智能以正确执行这种转换。)
3. 假如你想定义你自己的类，然后把它用作一个迭代器，在Python 3里，你可以通过定义特殊方法 `__next__()` 来实现。
4. 如果你定义的类里刚好有一个 `next()` ，它使用一个或者多个参数， `2to3` 执行的时候不会动它。这个类不能被当作迭代器使用，因为它的 `next()` 方法带有参数。
5. 这一个有些复杂。如果你恰好有一个叫做 `next` 的本地变量，在Python 3里它的优先级会高于全局函数 `next()` 。在这种情况下，你需要调用迭代器的特别方法 `__next__()` 来获取序列里的下一个元素。(或者，你也可以重构代码以使这个本地变量的名字不叫 `next` ，但是2to3不会为你做这件事。)

全局函数 `filter()`

在Python 2里， `filter()` 方法返回一个列表，这个列表是通过一个返回值为 `True` 或者 `False` 的函数来检测序列里的每一项得到的。在Python 3里， `filter()` 函数返回一个迭代器，不再是列表。

Notes	Python 2	Python 3
①	`filter(a_function, a_sequence)`	`list(filter(a_function, a_sequ`
②	`list(filter(a_function, a_sequence))`	*no change*
③	`filter(None, a_sequence)`	`[i for i in a_sequence if i]`
④	`for i in filter(None, a_sequence):`	*no change*
⑤	`[i for i in filter(a_function, a_sequence)]`	*no change*

1. 最简单的情况下，`2to3`会用一个`list()`函数来包装`filter()`，`list()`函数会遍历它的参数然后返回一个列表。
2. 然而，如果`filter()`调用已经被`list()`包裹，`2to3`不会再做处理，因为这种情况下`filter()`的返回值是否是一个迭代器是无关紧要的。
3. 为了处理`filter(None, ...)`这种特殊的语法，`2to3`会将这种调用从语法上等价地转换为列表解析。
4. 由于`for`循环会遍历整个序列，所以没有必要再做修改。
5. 与上面相同，不需要做修改，因为列表解析会遍历整个序列，即使`filter()`返回一个迭代器，它仍能像以前的`filter()`返回列表那样正常工作。

全局函数`map()`

跟`filter()`作的改变一样，`map()`函数现在返回一个迭代器。(在Python 2里，它返回一个列表。)

Notes	Python 2	Python 3
①	`map(a_function, 'PapayaWhip')`	`list(map(a_function, 'PapayaWhip'`
②	`map(None, 'PapayaWhip')`	`list('PapayaWhip')`
③	`map(lambda x: x+1, range(42))`	`[x+1 for x in range(42)]`
④	`for i in map(a_function, a_sequence):`	*no change*
⑤	`[i for i in map(a_function, a_sequence)]`	*no change*

1. 类似对`filter()`的处理，在最简单的情况下，`2to3`会用一个`list()`函数来包装`map()`调用。
2. 对于特殊的`map(None, ...)`语法，跟`filter(None, ...)`类似，`2to3`会将其转换成一个使用`list()`的等价调用
3. 如果`map()`的第一个参数是一个lambda函数，`2to3`会将其等价地转换成列表解析。
4. 对于会遍历整个序列的`for`循环，不需要做改变。
5. 再一次地，这里不需要做修改，因为列表解析会遍历整个序列，即使`map()`的返回值是迭代器而不是列表它也能正常工作。

全局函数`reduce()`

在Python 3里，`reduce()`函数已经被从全局名字空间里移除了，它现在被放置在`fucntools`模块里。

Notes	Python 2	Python 3
	`reduce(a, b, c)`	`from functools import reduce` `reduce(a, b, c)`

全局函数 `apply()`

Python 2有一个叫做 `apply()` 的全局函数，它使用一个函数 `f` 和一个列表 `[a, b, c]` 作为参数，返回值是 `f(a, b, c)` 。你也可以通过直接调用这个函数，在列表前添加一个星号(*)作为参数传递给它来完成同样的事情。在Python 3里， `apply()` 函数不再存在了；必须使用星号标记法。

Notes	Python 2	
①	`apply(a_function, a_list_of_args)`	`a_function(*`
②	`apply(a_function, a_list_of_args, a_dictionary_of_named_args)`	`a_function(*`
③	`apply(a_function, a_list_of_args + z)`	`a_function(*`
④	`apply(aModule.a_function, a_list_of_args)`	`aModule.a_fu`

1. 最简单的形式，可以通过在参数列表(就像 `[a, b, c]` 一样)前添加一个星号来调用函数。这跟Python 2里的 `apply()` 函数是等价的。
2. 在Python 2里， `apply()` 函数实际上可以带3个参数：一个函数，一个参数列表，一个字典命名参数(dictionary of named arguments)。在Python 3里，你可以通过在参数列表前添加一个星号(`*`)，在字典命名参数前添加两个星号(`**`)来达到同样的效果。
3. 运算符 `+` 在这里用作连接列表的功能，它的优先级高于运算符 `*` ，所以没有必要在 `a_list_of_args + z` 周围添加额外的括号。
4. `2to3` 脚本足够智能来转换复杂的 `apply()` 调用，包括调用导入模块里的函数。

全局函数 `intern()`

在Python 2里，你可以用 `intern()` 函数作用在一个字符串上来限定(intern)它以达到性能优化。在Python 3里， `intern()` 函数被转移到 `sys` 模块里了。

Notes	Python 2	Python 3
	`intern(aString)`	`sys.intern(aString)`

`exec` 语句

就像 `print` 语句在Python 3里变成了一个函数一样， `exec` 语句也是这样的。 `exec()` 函数使用一个包含任意Python代码的字符串作为参数，然后就像执行语句或者表达式一样执行它。 `exec()` 跟 `eval()` 是相似的，但是 `exec()` 更加强大并更具有技巧性。 `eval()` 函数只能执行单独一条表达式，但是 `` `exec `` ()`能够执行多条语句，导入(import)，函数声明 — 实际上整个Python程序的字符串表示也可以。

Notes	Python 2	
①	`exec codeString`	`exec(codeString)`
②	`exec codeString in a_global_namespace`	`exec(codeString,`
③	`exec codeString in a_global_namespace, a_local_namespace`	`exec(codeString,`

1. 在最简单的形式下，因为 `exec()` 现在是一个函数，而不是语句， `2to3` 会把这个字符串形式的代码用括号围起来。
2. Python 2里的 `exec` 语句可以指定名字空间，代码将在这个由全局对象组成的私有空间里执行。Python 3也有这样的功能；你只需要把这个名字空间作为第二个参数传递给 `exec()` 函数。
3. 更加神奇的是，Python 2里的 `exec` 语句还可以指定一个本地名字空间(比如一个函数里声明的变量)。在Python 3里， `exec()` 函数也有这样的功能。

`execfile` 语句

就像以前的 `exec` 语句，Python 2里的 `execfile` 语句也可以像执行Python代码那样使用字符串。不同的是 `exec` 使用字符串，而 `execfile` 则使用文件。在Python 3里， `execfile` 语句已经被去掉了。如果你真的想要执行一个文件里的Python代码(但是你不想导入它)，你可以通过打开这个文件，读取它的内容，然后调用 `compile()` 全局函数强制Python解释器编译代码，然后调用新的 `exec()` 函数。

Notes	Python 2	Python 3
	`execfile('a_filename')`	`exec(compile(open('a_filename').read(), 'a_filename`

`repr` (反引号)

在Python 2里，为了得到一个任意对象的字符串表示，有一种把对象包装在反引号里(比如 `x`)的特殊语法。在Python 3里，这种能力仍然存在，但是你不能再使用反引号获得这种字符串表示了。你需要使用全局函数 `repr()` 。

Notes	Python 2	Python 3
①	`` `x` ``	`repr(x)`
②	`` `'PapayaWhip' + `2`` ` ``	`repr('PapayaWhip' + repr(2))`

1. 记住， `x` 可以是任何东西 — 一个类，函数，模块，基本数据类型，等等。 `repr()` 函数可以使用任何类型的参数。
2. 在Python 2里，反引号可以嵌套，导致了这种令人费解的(但是有效的)表达式。 `2to3` 足够智能以将这种嵌套调用转换到 `repr()` 函数。

`try...except` 语句

从Python 2到Python 3，捕获异常的语法有些许变化。

Notes	Python 2	Python 3
①	[1]	[2]
②	[3]	[4]
③	[5]	*no change*
④	[6]	*no change*

[1]:

```
try:
    import mymodule
except ImportError, e
    pass
```

[2]:

```
try:
    import mymodule
except ImportError as e:
    pass
```

[3]:

```
try:
    import mymodule
except (RuntimeError, ImportError), e
    pass
```

[4]:

```
try:
    import mymodule
except (RuntimeError, ImportError) as e:
    pass
```

[5]:

```
try:
    import mymodule
except ImportError:
    pass
```

[6]:

```
try:
    import mymodule
except:
    pass
```

1. 相对于Python 2里在异常类型后添加逗号，Python 3使用了一个新的关键字， `as` 。
2. 关键字 `as` 也可以用在一次捕获多种类型异常的情况下。
3. 如果你捕获到一个异常，但是并不在意访问异常对象本身，Python 2和Python 3的语法是一样的。
4. 类似地，如果你使用一个保险方法(fallback)来捕获所有异常，Python 2和Python 3的语法是一样的。

> ☞在导入模块(或者其他大多数情况)的时候，你绝对不应该使用这种方法(指以上的fallback)。不然的话，程序可能会捕获到像 `KeyboardInterrupt` (如果用户按 `Ctrl-C` 来中断程序)这样的异常，从而使调试变得更加困难。

`raise` 语句

Python 3里，抛出自定义异常的语法有细微的变化。

Notes	Python 2	
①	`raise MyException`	*unchanged*
②	`raise MyException, 'error message'`	`raise MyException('error r`
③	`raise MyException, 'error message', a_traceback`	`raise MyException('error r`
④	`raise 'error message'`	*unsupported*

1. 抛出不带用户自定义错误信息的异常，这种最简单的形式下，语法没有改变。
2. 当你想要抛出一个带用户自定义错误信息的异常时，改变就显而易见了。Python 2用一个逗号来分隔异常类和错误信息；Python 3把错误信息作为参数传递给异常类。
3. Python 2支持一种更加复杂的语法来抛出一个带用户自定义回溯(stack trace，堆栈追踪)的异常。在Python 3里你也可以这样做，但是语法完全不同。
4. 在Python 2里，你可以抛出一个不带异常类的异常，仅仅只有一个异常信息。在Python 3里，这种形式不再被支持。 `2to3` 将会警告你它不能自动修复这种语法。

生成器的 `throw` 方法

在Python 2里，生成器有一个 `throw()` 方法。调用 `a_generator.throw()` 会在生成器被暂停的时候抛出一个异常，然后返回由生成器函数获取的下一个值。在Python 3里，这种功能仍然可用，但是语法上有一点不同。

Notes	Python 2	Pytho
①	`a_generator.throw(MyException)`	*no change*
②	`a_generator.throw(MyException, 'error message')`	`a_generator.throw(MyExcept`
③	`a_generator.throw('error message')`	*unsupported*

1. 最简单的形式下，生成器抛出不带用户自定义错误信息的异常。这种情况下，从Python 2到Python 3语法上没有变化 。
2. 如果生成器抛出一个带用户自定义错误信息的异常，你需要将这个错误信息字符串(error string)传递给异常类来以实例化它。
3. Python 2还支持抛出只有异常信息的异常。Python 3不支持这种语法，并且 `2to3` 会显示一个警告信息，告诉你需要手动地来修复这处代码。

全局函数 `xrange()`

在Python 2里，有两种方法来获得一定范围内的数字： `range()` ，它返回一个列表，还有 `range()` ，它返回一个迭代器。在Python 3里， `range()` 返回迭代器， `xrange()` 不再存在了。

Notes	Python 2	Python 3
①	`xrange(10)`	`range(10)`
②	`a_list = range(10)`	`a_list = list(range(10))`
③	`[i for i in xrange(10)]`	`[i for i in range(10)]`
④	`for i in range(10):`	*no change*
⑤	`sum(range(10))`	*no change*

1. 在最简单的情况下， `2to3` 会简单地把 `xrange()` 转换为 `range()` 。
2. 如果你的Python 2代码使用 `range()` ， `2to3` 不知道你是否需要一个列表，或者是否一个迭代器也行。出于谨慎， `2to3` 可能会报错，然后使用 `list()` 把 `range()` 的返回值强制转换为列表类型。
3. 如果在列表解析里有 `xrange()` 函数，就没有必要将其返回值转换为一个列表，因为列表解析对迭代器同样有效。
4. 类似的， `for` 循环也能作用于迭代器，所以这里也没有改变任何东西。
5. 函数 `sum()` 能作用于迭代器，所以 `2to3` 也没有在这里做出修改。就像返回值为视图(view)而不再是列表的字典类方法一样，这同样适用于 `min()` ， `max()` ， `sum()` ，list()， `tuple()` ， `set()` ， `sorted()` ， `any()` ， `all()` 。

全局函数 `raw_input()` 和 `input()`

Python 2有两个全局函数，用来在命令行请求用户输入。第一个叫做 `input()` ，它等待用户输入一个Python表达式(然后返回结果)。第二个叫做 `raw_input()` ，用户输入什么它就返回什么。这让初学者非常困惑，并且这被广泛地看作是Python语言的一个“肉赘”(wart)。Python 3通过重命名 `raw_input()` 为 `input()` ，从而切掉了这个肉赘，所以现在的 `input()` 就像每个人最初期待的那样工作。

Notes	Python 2	Python 3
①	`raw_input()`	`input()`
②	`raw_input('prompt')`	`input('prompt')`
③	`input()`	`eval(input())`

1. 最简单的形式， `raw_input()` 被替换成 `input()` 。
2. 在Python 2里， `raw_input()` 函数可以指定一个提示符作为参数。Python 3里保留了这个功能。
3. 如果你真的想要请求用户输入一个Python表达式，计算结果，可以通过调用 `input()` 函数然后把返回值传递给 `eval()` 。

函数属性 `func_*`

在Python 2里，函数的里的代码可以访问到函数本身的特殊属性。在Python 3里，为了一致性，这些特殊属性被重新命名了。

Notes	Python 2	Python 3
①	`a_function.func_name`	`a_function.__name__`
②	`a_function.func_doc`	`a_function.__doc__`
③	`a_function.func_defaults`	`a_function.__defaults__`
④	`a_function.func_dict`	`a_function.__dict__`
⑤	`a_function.func_closure`	`a_function.__closure__`
⑥	`a_function.func_globals`	`a_function.__globals__`
⑦	`a_function.func_code`	`a_function.__code__`

1. `__name__` 属性(原 `func_name`)包含了函数的名字。
2. `__doc__` 属性(原 `funcdoc`)包含了你在函数源代码里定义的文档字符串(*docstring*)
3. `__defaults__` 属性(原 `func_defaults`)是一个保存参数默认值的元组。
4. `__dict__` 属性(原 `func_dict`)是一个支持任意函数属性的名字空间。
5. `__closure__` 属性(原 `func_closure`)是一个由cell对象组成的元组，它包含了函数对自由变量(free variable)的绑定。
6. `__globals__` 属性(原 `func_globals`)是一个对模块全局名字空间的引用，函数本身在这个

名字空间里被定义。
7. `__code__` 属性(原 `func_code`)是一个代码对象，表示编译后的函数体。

I/O方法 `xreadlines()`

在Python 2里，文件对象有一个 `xreadlines()` 方法，它返回一个迭代器，一次读取文件的一行。这在 `for` 循环中尤其有用。事实上，后来的Python 2版本给文件对象本身添加了这样的功能。

在Python 3里， `xreadlines()` 方法不再可用了。 `2to3` 可以解决简单的情况，但是一些边缘案例则需要人工介入。

Notes	Python 2	Python 3
①	`for line in a_file.xreadlines():`	`for line in a_file:`
②	`for line in a_file.xreadlines(5):`	*no change (broken)*

1. 如果你以前调用没有参数的 `xreadlines()` ， `2to3` 会把它转换成文件对象本身。在Python 3里，这种转换后的代码可以完成前同样的工作：一次读取文件的一行，然后执行 `for` 循环的循环体。
2. 如果你以前使用一个参数(每次读取的行数)调用 `xreadlines()` ， `2to3` 不能为你完成从Python 2到Python 3的转换，你的代码会以这样的方式失败： `AttributeError: '_io.TextIOWrapper' object has no attribute 'xreadlines'` 。你可以手工的把 `xreadlines()` 改成 `readlines()` 以使代码能在Python 3下工作。(readline()方法在Python 3里返回迭代器，所以它跟Python 2里的 `xreadlines()` 效率是不相上下的。)

☃

使用元组而非多个参数的 `lambda` 函数

在Python 2里，你可以定义匿名 `lambda` 函数(anonymous `lambda` function)，通过指定作为参数的元组的元素个数，使这个函数实际上能够接收多个参数。事实上，Python 2的解释器把这个元组“解开”(unpack)成命名参数(named arguments)，然后你可以在 `lambda` 函数里引用它们(通过名字)。在Python 3里，你仍然可以传递一个元组作为 `lambda` 函数的参数，但是Python解释器不会把它解析成命名参数。你需要通过位置索引(positional index)来引用每个参数。

Notes	Python 2	Python 3
①	`lambda (x,): x + f(x)`	`lambda x1: x1[0] + f(x1[0])`
②	`lambda (x, y): x + f(y)`	`lambda x_y: x_y[0] + f(x_y[1])`
③	`lambda (x, (y, z)): x + y + z`	`lambda x_y_z: x_y_z[0] + x_y_z[1][0] + x_y_z`
④	`lambda x, y, z: x + y + z`	*unchanged*

1. 如果你已经定义了一个 `lambda` 函数，它使用包含一个元素的元组作为参数，在Python 3里，它会被转换成一个包含到 `x1[0]` 的引用的 `lambda` 函数。 `x1` 是 `2to3` 脚本基于原来元组里的命名参数自动生成的。
2. 使用含有两个元素的元组 `(x, y)` 作为参数的 `lambda` 函数被转换为 `x_y` ，它有两个位置参数，即 `x_y[0]` 和 `x_y[1]` 。
3. `2to3` 脚本甚至可以处理使用嵌套命名参数的元组作为参数的 `lambda` 函数。产生的结果代码有点难以阅读，但是它在Python 3下跟原来的代码在Python 2下的效果是一样的。
4. 你可以定义使用多个参数的 `lambda` 函数。如果没有括号包围在参数周围，Python 2会把它当作一个包含多个参数的 `lambda` 函数；在这个 `lambda` 函数体里，你通过名字引用这些参数，就像在其他类型的函数里所做的一样。这种语法在Python 3里仍然有效。

特殊的方法属性

在Python 2里，类方法可以访问到定义他们的类对象(class object)，也能访问方法对象(method object)本身。 `im_self` 是类的实例对象； `im_func` 是函数对象， `im_class` 是类本身。在Python 3里，这些属性被重新命名，以遵循其他属性的命名约定。

Notes	Python 2	Python 3
	`aClassInstance.aClassMethod.im_func`	`aClassInstance.aClassMethod.__func__`
	`aClassInstance.aClassMethod.im_self`	`aClassInstance.aClassMethod.__self__`
	`aClassInstance.aClassMethod.im_class`	`aClassInstance.aClassMethod.__self__`

`__nonzero__` 特殊方法

在Python 2里，你可以创建自己的类，并使他们能够在布尔上下文(boolean context)中使用。举例来说，你可以实例化这个类，并把这个实例对象用在一个 `if` 语句中。为了实现这个目的，你定义一个特别的 `__nonzero__()` 方法，它的返回值为 `True` 或者 `False` ，当实例对象处在布尔上下文中的时候这个方法就会被调用 。在Python 3里，你仍然可以完成同样的功能，但是这个特殊方法的名字变成了 `__bool__()` 。

Notes	Python 2	Python 3
①	[1]	[2]
②	[3]	*no change*

[1]:

```
class A:
    def __nonzero__(self):
        pass
```

[2]:

```
class A:
    def __bool__(self):
        pass
```

[3]:

```
class A:
    def __nonzero__(self, x, y):
        pass
```

1. 当在布尔上下文使用一个类对象时，Python 3会调用 `__bool__()` ，而非 `__nonzero__()` 。
2. 然而，如果你有定义了一个使用两个参数的 `__nonzero__()` 方法， `2to3` 脚本会假设你定义的这个方法有其他用处，因此不会对代码做修改。

八进制类型

在Python 2和Python 3之间，定义八进制(octal)数的语法有轻微的改变。

Notes	Python 2	Python 3
	`x = 0755`	`x = 0o755`

`sys.maxint`

由于长整型和整型被整合在一起了， `sys.maxint` 常量不再精确。但是因为这个值对于检测特定平台的能力还是有用处的，所以它被Python 3保留，并且重命名为 `sys.maxsize` 。

Notes	Python 2	Python 3
①	`from sys import maxint`	`from sys import maxsize`
②	`a_function(sys.maxint)`	`a_function(sys.maxsize)`

1. `maxint` 变成了 `maxsize` 。
2. 所有的 `sys.maxint` 都变成了 `sys.maxsize` 。

全局函数 `callable()`

在Python 2里，你可以使用全局函数 `callable()` 来检查一个对象是否可调用(callable，比如函数)。在Python 3里，这个全局函数被取消了。为了检查一个对象是否可调用，可以检查特殊方法 `__call__()` 的存在性。

Notes	Python 2	Python 3
	`callable(anything)`	`hasattr(anything, '__call__')`

全局函数 `zip()`

在Python 2里，全局函数 `zip()` 可以使用任意多个序列作为参数，它返回一个由元组构成的列表。第一个元组包含了每个序列的第一个元素；第二个元组包含了每个序列的第二个元素；依次递推下去。在Python 3里， `zip()` 返回一个迭代器，而非列表。

Notes	Python 2	Python 3
①	`zip(a, b, c)`	`list(zip(a, b, c))`
②	`d.join(zip(a, b, c))`	*no change*

1. 最简单的形式，你可以通过调用 `list()` 函数包装 `zip()` 的返回值来恢复 `zip()` 函数以前的功能， `list()` 函数会遍历这个 `zip()` 函数返回的迭代器，然后返回结果的列表表示。
2. 在已经会遍历序列所有元素的上下文环境里(比如这里对 `join()` 方法的调用)， `zip()` 返回的迭代器能够正常工作。 `2to3` 脚本会检测到这些情况，不会对你的代码作出改变。

`StandardError` 异常

在Python 2里， `StandardError` 是除了 `StopIteration` ， `GeneratorExit` ， `KeyboardInterrupt` ， `SystemExit` 之外所有其他内置异常的基类。在Python 3里， `StandardError` 已经被取消了；使用 `Exception` 替代。

Notes	Python 2	Python 3
	`x = StandardError()`	`x = Exception()`
	`x = StandardError(a, b, c)`	`x = Exception(a, b, c)`

`types` 模块中的常量

`types` 模块里各种各样的常量能帮助你决定一个对象的类型。在Python 2里，它包含了代表所有基本数据类型的常量，如 `dict` 和 `int` 。在Python 3里，这些常量被已经取消了。只需要使用基础类型的名字来替代。

Notes	Python 2	Python 3
	`types.UnicodeType`	`str`
	`types.StringType`	`bytes`
	`types.DictType`	`dict`
	`types.IntType`	`int`
	`types.LongType`	`int`
	`types.ListType`	`list`
	`types.NoneType`	`type(None)`
	`types.BooleanType`	`bool`
	`types.BufferType`	`memoryview`
	`types.ClassType`	`type`
	`types.ComplexType`	`complex`
	`types.EllipsisType`	`type(Ellipsis)`
	`types.FloatType`	`float`
	`types.ObjectType`	`object`
	`types.NotImplementedType`	`type(NotImplemented)`
	`types.SliceType`	`slice`
	`types.TupleType`	`tuple`
	`types.TypeType`	`type`
	`types.XRangeType`	`range`

> ☞ `types.StringType` 被映射为 `bytes` ，而非 `str` ，因为Python 2里的“string”(非Unicode编码的字符串，即普通字符串)事实上只是一些使用某种字符编码的字节序列(a sequence of bytes)。

全局函数 `isinstance()`

`isinstance()` 函数检查一个对象是否是一个特定类(class)或者类型(type)的实例。在Python 2里，你可以传递一个由类型(types)构成的元组给 `isinstance()` ，如果该对象是元组里的任意一种类型，函数返回 `True` 。在Python 3里，你依然可以这样做，但是不推荐使用把一种类型作为参数传递两次。

Notes	Python 2	Python 3
	`isinstance(x, (int, float, int))`	`isinstance(x, (int, float))`

`basestring` 数据类型

Python 2有两种字符串类型：Unicode编码的字符串和非Unicode编码的字符串。但是其实还有另外 一种类型，即 `basestring` 。它是一个抽象数据类型，是 `str` 和 `unicode` 类型的超类(superclass)。它不能被直接调用或者实例化，但是你可以把它作为 `isinstance()` 的参数来检测一个对象是否是一个Unicode字符串或者非Unicode字符串。在Python 3里，只有一种字符串类型，所以 `basestring` 就没有必要再存在了。

Notes	Python 2	Python 3
	`isinstance(x, basestring)`	`isinstance(x, str)`

`itertools` 模块

Python 2.3引入了 `itertools` 模块，它定义了全局函数 `zip()` ， `map()` ， `filter()` 的变体(variant)，这些变体的返回类型为迭代器，而非列表。在Python 3里，由于这些全局函数的返回类型本来就是迭代器，所以这些 `itertools` 里的这些变体函数就被取消了。(在 `itertools` 模块里仍然还有许多其他的有用的函数，而不仅仅是以上列出的这些。)

Notes	Python 2	Python 3
①	`itertools.izip(a, b)`	`zip(a, b)`
②	`itertools.imap(a, b)`	`map(a, b)`
③	`itertools.ifilter(a, b)`	`filter(a, b)`
④	`from itertools import imap, izip, foo`	`from itertools import foo`

1. 使用全局的 `zip()` 函数，而非 `itertools.izip()` 。
2. 使用 `map()` 而非 `itertools.imap()` 。
3. `itertools.ifilter()` 变成了 `filter()` 。
4. `itertools` 模块在Python 3里仍然存在，它只是不再包含那些已经转移到全局名字空间的函数。 `2to3` 脚本能够足够智能地去移除那些不再有用的导入语句，同时保持其他的导入语句的完整性。

`sys.exc_type` ， `sys.exc_value` ， `sys.exc_traceback`

处理异常的时候，在 `sys` 模块里有三个你可以访问的变量： `sys.exc_type`， sys.exc_value， `sys.exc_traceback` 。(实际上这些在Python 1的时代就有。)从Python 1.5开始，由于新出的 `sys.exc_info` ，不再推荐使用这三个变量了，这是一个包含所有以上三个元素的元组。在Python 3里，这三个变量终于不再存在了；这意味着，你必须使用 `sys.exc_info` 。``

Notes	Python 2	Python 3
	`sys.exc_type`	`sys.exc_info()[0]`
	`sys.exc_value`	`sys.exc_info()[1]`
	`sys.exc_traceback`	`sys.exc_info()[2]`

对元组的列表解析

在Python 2里，如果你需要编写一个遍历元组的列表解析，你不需要在元组值的周围加上括号。在Python 3里，这些括号是必需的。

Notes	Python 2	Python 3
	`[i for i in 1, 2]`	`[i for i in (1, 2)]`

`os.getcwdu()` 函数

Python 2有一个叫做 `os.getcwd()` 的函数，它将当前的工作目录作为一个(非Unicode编码的)字符串返回。由于现代的文件系统能够处理能何字符编码的目录名，Python 2.3引入了 `os.getcwdu()` 函数。 `os.getcwdu()` 函数把当前工作目录用Unicode编码的字符串返回。在Python 3里，由于只有一种字符串类型(Unicode类型的)，所以你只需要 `os.getcwd()` 就可以了。

Notes	Python 2	Python 3
	`os.getcwdu()`	`os.getcwd()`

元类(metaclass)

在Python 2里，你可以通过在类的声明中定义 `metaclass` 参数，或者定义一个特殊的类级别的(class-level) `__metaclass__` 属性，来创建元类。在Python 3里， `__metaclass__` 属性已经被取消了。

Notes	Python 2	Python 3
①	[1]	*unchanged*
②	[2]	[3]
③	[4]	[5]

[1]:

```
class C(metaclass=PapayaMeta):
    pass
```

[2]:

```
class Whip:
    __metaclass__ = PapayaMeta
```

[3]:

```
class Whip(metaclass=PapayaMeta):
    pass
```

[4]:

```
class C(Whipper, Beater):
    __metaclass__ = PapayaMeta
```

[5]:

```
class C(Whipper, Beater, metaclass=PapayaMeta):
    pass
```

1. 在声明类的时候声明 `metaclass` 参数，这在Python 2和Python 3里都有效，它们是一样的。
2. 在类的定义里声明 `__metaclass__` 属性在Python 2里有效，但是在Python 3里不再有效。
3. `2to3` 能够构建一个有效的类声明，即使这个类继承自多个父类。

关于代码风格

以下所列的“修补”(fixes)实质上并不算真正的修补。意思就是，他们只是代码的风格上的事情，而不涉及到代码的本质。但是Python的开发者们在使得代码风格尽可能一致方面非常有兴趣(have a vested interest)。为此，有一个专门o描述Python代码风格的官方指导手册 — 细致到能使人痛苦 — 都是一些你不太可能关心的在各种各样的细节上的挑剔。鉴于 `2to3` 为转换代码提供了一个这么好的条件，脚本的作者们添加了一些可选的特性以使你的代码更具可读性。

`set()` 字面值(literal)(显式的)

在Python 2城，定义一个字面值集合(literal set)的唯一方法就是调用 `set(a_sequence)` 。在Python 3里这仍然有效，但是使用新的标注记号(literal notation)：大括号({})是一种更清晰的方法。这种方法除了空集以外都有效，因为字典也用大括号标记，所以 `{}` 表示一个空的字典，而不是一个空集。

> ☞ `2to3` 脚本默认不会修复 `set()` 字面值。为了开启这个功能，在命令行调用 `2to3` 的时候指定 `-f set_literal` 参数。

Notes	Before	After
	`set([1, 2, 3])`	`{1, 2, 3}`
	`set((1, 2, 3))`	`{1, 2, 3}`
	`set([i for i in a_sequence])`	`{i for i in a_sequence}`

全局函数 `buffer()` (显式的)

用C实现的Python对象可以导出一个“缓冲区接口”(buffer interface)，它允许其他的Python代码直接读写一块内存。(这听起来很强大，它也同样可怕。)在Python 3里，`buffer()` 被重新命名为 `memoryview()` 。(实际的修改更加复杂，但是你几乎可以忽略掉这些不同之处。)

> ☞ `2to3` 脚本默认不会修复 `buffer()` 函数。为了开启这个功能，在命令行调用 `2to3` 的时候指定 `-f buffer` 参数。

Notes	Before	After
	`x = buffer(y)`	`x = memoryview(y)`

逗号周围的空格(显式的)

尽管Python对用于缩进和凸出(indenting and outdenting)的空格要求很严格，但是对于空格在其他方面的使用Python还是很自由的。在列表，元组，集合和字典里，空格可以出现在逗号的前面或者后面，这不会有什么坏影响。但是，Python代码风格指导手册上指出，逗号前不能有空格，逗号后应该包含一个空格。尽管这纯粹只是一个美观上的考量(代码仍然可以正常工作，在Python 2和Python 3里都可以)，但是 `2to3` 脚本可以依据手册上的标准为你完成这个修复。

> ☞ `2to3` 脚本默认不会修复逗号周围的空格。为了开启这个功能，在命令行调用 `2to3` 的时候指定 `-f wscomma` 参数。

Notes	Before	After
	`a ,b`	`a, b`
	`{a :b}`	`{a: b}`

惯例(Common idioms)(显式的)

在Python社区里建立起来了许多惯例。有一些比如 `while 1:` loop，它可以追溯到Python 1。(Python直到Python 2.3才有真正意义上的布尔类型，所以开发者以前使用 `1` 和 `0` 替代。)当代的Python程序员应该锻炼他们的大脑以使用这些惯例的现代版。

> ☞ `2to3` 脚本默认不会为这些惯例做修复。为了开启这个功能，在命令行调用 `2to3` 的时候指定 `-f idioms` 参数。

Notes	Before	After
	[1]	[2]
	`type(x) == T`	`isinstance(x, T)`
	`type(x) is T`	`isinstance(x, T)`
	[3]	[4]

[1]:

```
while 1:
    do_stuff()
```

[2]:

```
while True:
    do_stuff()
```

[3]:

```
a_list = list(a_sequence)
a_list.sort()
do_stuff(a_list)
```

[4]:

```
a_list = sorted(a_sequence)
do_stuff(a_list)
```

Chapter B 特殊方法名称

> " My specialty is being right when other people are wrong. " — George Bernard Shaw

深入

在本书其它几处，我们已经见识过一些特殊方法——即在使用某些语法时 Python 所调用的“神奇”方法。使用特殊方法，类用起来如同序列、字典、函数、迭代器，或甚至像个数字！本附录为我们已经见过特殊方法提供了参考，并对一些更加深奥的特殊方法进行了简要介绍。

基础知识

如果曾阅读 《类的简介》一章，你可能已经见识过了最常见的特殊方法： `__init__()` 方法。盖章结束时，我写的类多数需要进行一些初始化工作。还有一些其它的基础特殊方法对调试自定义类也特别有用。

序号	目的	所编写代码	Python 实际调用
①	初始化一个实例	`x = MyClass()`	`x.__init__()`
②	字符串的“官方”表现形式	`repr(x)`	`x.__repr__()`
③	字符串的“非正式”值	`str(x)`	`x.__str__()`
④	字节数组的“非正式”值	`bytes(x)`	`x.__bytes__()`
⑤	格式化字符串的值	`format(x,` format_spec `)`	`x.__format__(` format_spec `)`

1. 对 `__init__()` 方法的调用发生在实例被创建 之后 。如果要控制实际创建进程，请使用 `__new__()` 方法。
2. 按照约定， `__repr__()` 方法所返回的字符串为合法的 Python 表达式。
3. 在调用 `print(x)` 的同时也调用了 `__str__()` 方法。
4. 由于 `bytes` 类型的引入而从 *Python 3* 开始出现。
5. 按照约定， `format_spec` 应当遵循 迷你语言格式规范【Format Specification Mini-Language】。Python 标准类库中的 `decimal.py` 提供了自己的 `__format__()` 方法。

行为方式与迭代器类似的类

在 《迭代器》一章中，我们已经学习了如何使用 `__iter__()` 和 `__next__()` 方法从零开始创建迭代器。

序号	目的	所编写代码	**Python** 实际调用
①	遍历某个序列	`iter(seq)`	`seq.__iter__()`
②	从迭代器中获取下一个值	`next(seq)`	`seq.__next__()`
③	按逆序创建一个迭代器	`reversed(seq)`	`seq.__reversed__()`

1. 无论何时创建迭代器都将调用 `__iter__()` 方法。这是用初始值对迭代器进行初始化的绝佳之处。
2. 无论何时从迭代器中获取下一个值都将调用 `__next__()` 方法。
3. `__reversed__()` 方法并不常用。它以一个现有序列为参数，并将该序列中所有元素从尾到头以逆序排列生成一个新的迭代器。

正如我们在 《迭代器》一章中看到的， `for` 循环也可用作迭代器。在下面的循环中：

```
for x in seq:
    print(x)
```

Python 3 将会调用 `seq.__iter__()` 以创建一个迭代器，然后对迭代器调用 `__next__()` 方法以获取 `x` 的每个值。当 `__next__()` 方法引发 `StopIteration` 例外时， `for` 循环正常结束。

计算属性

序号	目的	所编写代码	Python 实际调用
①	获取一个计算属性（无条件的）	`x.my_property`	`x.__getattribute__(` 'my_property' `)`
---	---	---	---
②	获取一个计算属性（后备）	`x.my_property`	`x.__getattr__(` 'my_property' `)`
---	---	---	---
③	设置某属性	`x.my_property = value`	`x.__setattr__(` 'my_property' `,` value `)`
---	---	---	---
④	删除某属性	`del x.my_property`	`x.__delattr__(` 'my_property' `)`
---	---	---	---
⑤	列出所有属性和方法	`dir(x)`	`x.__dir__()`
---	---	---	---

1. 如果某个类定义了 `__getattribute__()` 方法，在 每次引用属性或方法名称时 Python 都调用它（特殊方法名称除外，因为那样将会导致讨厌的无限循环）。
2. 如果某个类定义了 `__getattr__()` 方法，Python 将只在正常的位置查询属性时才会调用它。如果实例 `x` 定义了属性 `color`， `x.color` 将 不会 调用 `x.__getattr__('color')`；而只会返回 `x.color` 已定义好的值。
3. 无论何时给属性赋值，都会调用 `__setattr__()` 方法。
4. 无论何时删除一个属性，都将调用 `__delattr__()` 方法。
5. 如果定义了 `__getattr__()` 或 `__getattribute__()` 方法， `__dir__()` 方法将非常有用。通常，调用 `dir(x)` 将只显示正常的属性和方法。如果 `__getattr()__` 方法动态处理 `color` 属性， `dir(x)` 将不会将 `color` 列为可用属性。可通过覆盖 `__dir__()` 方法允许将 `color` 列为可用属性，对于想使用你的类但却不想深入其内部的人来说，该方法非常有益。

`__getattr__()` 和 `__getattribute__()` 方法的区别非常细微，但非常重要。可以用两个例子来解释一下：

```
class Dynamo:
    def __getattr__(self, key):

            return 'PapayaWhip'
        else:

>>> dyn = Dynamo()

'PapayaWhip'
>>> dyn.color = 'LemonChiffon'

'LemonChiffon'
```

1. 属性名称以字符串的形式传入 `__getattr()__` 方法。如果名称为 `'color'`，该方法返回一个值。（在此情况下，它只是一个硬编码的字符串，但可以正常地进行某些计算并返回结果。）
2. 如果属性名称未知， `__getattr()__` 方法必须引发一个 `AttributeError` 例外，否则在访问未定义属性时，代码将只会默默地失败。（从技术角度而言，如果方法不引发例外或显式地返回一个值，它将返回 `None` ——Python 的空值。这意味着 所有 未显式定义的属性将为 `None`，几乎可以肯定这不是你想看到的。）
3. `dyn` 实例没有名为 `color` 的属性，因此在提供计算值时将调用 `__getattr__()` 。
4. 在显式地设置 `dyn.color` 之后，将不再为提供 `dyn.color` 的值而调用 `__getattr__()` 方法，因为 `dyn.color` 已在该实例中定义。

另一方面， `__getattribute__()` 方法是绝对的、无条件的。

```
class SuperDynamo:
    def __getattribute__(self, key):
        if key == 'color':
            return 'PapayaWhip'
        else:
            raise AttributeError

>>> dyn = SuperDynamo()

'PapayaWhip'
>>> dyn.color = 'LemonChiffon'

'PapayaWhip'
```

1. 在获取 `dyn.color` 的值时将调用 `__getattribute__()` 方法。
2. 即便已经显式地设置 `dyn.color`，在获取 `dyn.color` 的值时, 仍将调用 `__getattribute__()` 方法。如果存在 `__getattribute__()` 方法，将在每次查找属性和方法时 无条件地调用 它，哪怕在创建实例之后已经显式地设置了属性。

> ☞ 如果定义了类的 `__getattribute__()` 方法，你可能还想定义一个 `__setattr__()` 方法，并在两者之间进行协同，以跟踪属性的值。否则，在创建实例之后所设置的值将会消失在黑洞中。

必须特别小心 `__getattribute__()` 方法，因为 Python 在查找类的方法名称时也将对其进行调用。

```
class Rastan:
    def __getattribute__(self, key):

    def swim(self):
        pass

>>> hero = Rastan()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __getattribute__
AttributeError
```

1. 该类定义了一个总是引发 `AttributeError` 例外的 `__getattribute__()` 方法。没有属性或方法的查询会成功。
2. 调用 `hero.swim()` 时，Python 将在 `Rastan` 类中查找 `swim()` 方法。该查找将执行整个 `__getattribute__()` 方法，因为所有的属性和方法查找都通过 `__getattribute__()` 方法。在此例中， `__getattribute__()` 方法引发 `AttributeError` 例外，因此该方法查找过程将会失败，而方法调用也将失败。

行为方式与函数类似的类

可以让类的实例变得可调用——就像函数可以调用一样——通过定义 `__call__()` 方法。

序号	目的	所编写代码	Python 实际调用
	像调用函数一样“调用”一个实例	`my_instance()`	`my_instance.__call__()`

`zipfile` 模块 通过该方式定义了一个可以使用给定密码解密 经加密 zip 文件的类。该 zip 解密 算法需要在解密的过程中保存状态。通过将解密器定义为类，使我们得以在 decryptor 类的单个实例中对该状态进行维护。状态在 `__init__()` 方法中进行初始化，如果文件 经加密 则进行更新。但由于该类像函数一样“可调用”，因此可以将实例作为 `map()` 函数的第一个参数传入，代码如下：

```
# excerpt from zipfile.py
class _ZipDecrypter:
.
.
.
    def __init__(self, pwd):

        self.key1 = 591751049
        self.key2 = 878082192
        for p in pwd:
            self._UpdateKeys(p)

        assert isinstance(c, int)
        k = self.key2 | 2
        c = c ^ (((k * (k^1)) >> 8) & 255)
        self._UpdateKeys(c)
        return c
.
.
.

bytes = zef_file.read(12)
```

1. `_ZipDecryptor` 类维护了以三个旋转密钥形式出现的状态，该状态稍后将在 `_UpdateKeys()` 方法中更新（此处未展示）。
2. 该类定义了一个 `__call__()` 方法，使得该类可像函数一样调用。在此例中， `__call__()` 对 zip 文件的单个字节进行解密，然后基于经解密的字节对旋转密码进行更新。
3. `zd` 是 `_ZipDecryptor` 类的一个实例。变量 `pwd` 被传入 `__init__()` 方法，并在其中被存储和用于首次旋转密码更新。
4. 给出 zip 文件的头 12 个字节，将这些字节映射给 `zd` 进行解密，实际上这将导致调用 `__call__()` 方法 12 次，也就是 更新内部状态并返回结果字节 12 次。

行为方式与序列类似的类

如果类作为一系列值的容器出现——也就是说如果对某个类来说，是否“包含”某值是件有意义的事情——那么它也许应该定义下面的特殊方法已，让它的行为方式与序列类似。

序号	目的	所编写代码	**Python** 实际调用
	序列的长度	`len(seq)`	`seq.__len__()`
	了解某序列是否包含特定的值	`x in seq`	`seq.__contains__( x )`

`cgi` 模块 在其 `FieldStorage` 类中使用了这些方法，该类用于表示提交给动态网页的所有表单字段或查询参数。

```
# A script which responds to http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()

  do_search()

# An excerpt from cgi.py that explains how that works
class FieldStorage:
.
.
.

        if self.list is None:
            raise TypeError('not indexable')
```

1. 一旦创建了 `cgi.FieldStorage` 类的实例，就可以使用 “ `in` ” 运算符来检查查询字符串中是否包含了某个特定参数。
2. 而 `__contains__()` 方法是令该魔法生效的主角。
3. 如果代码为 `if 'q' in fs` ，Python 将在 `fs` 对象中查找 `__contains__()` 方法，而该方法在 `cgi.py` 中已经定义。 `'q'` 的值被当作 `key` 参数传入 `__contains__()` 方法。
4. 同样的 `FieldStorage` 类还支持返回其长度，因此可以编写代码 `len(` fs `)` 而其将调用 `FieldStorage` 的 `__len__()` 方法，并返回其识别的查询参数个数。
5. `self.keys()` 方法检查 `self.list is None` 是否为真值，因此 `__len__` 方法无需重复该错误检查。

行为方式与字典类似的类

在前一节的基础上稍作拓展，就不仅可以对 “ `in` ” 运算符和 `len()` 函数进行响应，还可像全功能字典一样根据键来返回值。

序号	目的	所编写代码	Python 实际调用
	通过键来获取值	`x[key]`	`x.__getitem__(` key `)`
	通过键来设置值	`x[key] = value`	`x.__setitem__(` key `,` value `)`
	删除一个键值对	`del x[key]`	`x.__delitem__(` key `)`
	为缺失键提供默认值	`x[nonexistent_key]`	`x.__missing__(` nonexistent_key `)`

`cgi` 模块 的 `FieldStorage` 类 同样定义了这些特殊方法，也就是说可以像下面这样编码：

```
# A script which responds to http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:

# An excerpt from cgi.py that shows how it works
class FieldStorage:
.
.
.

        if self.list is None:
            raise TypeError('not indexable')
        found = []
        for item in self.list:
            if item.name == key: found.append(item)
        if not found:
            raise KeyError(key)
        if len(found) == 1:
            return found[0]
        else:
            return found
```

1. `fs` 对象是 `cgi.FieldStorage` 类的一个实例，但仍然可以像 `fs['q']` 这样估算表达式。
2. `fs['q']` 将 `key` 参数设置为 `'q'` 来调用 `__getitem__()` 方法。然后它将在其内部维护的查询参数列表 (`self.list`) 中查找一个 `.name` 与给定键相符的字典项。

行为方式与数值类似的类

使用适当的特殊方法，可以将类的行为方式定义为与数字相仿。也就是说，可以进行相加、相减，并进行其它数学运算。这就是 分数 的实现方式—— `Fraction` 类实现了这些特殊方法，然后就可以进行下列运算了：

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> x / 3
Fraction(1, 9)
```

以下是实现“类数字”类的完整特殊方法清单：

序号	目的	所编写代码	Python 实际调用
	加法	x + y	x.__add__(y)
	减法	x - y	x.__sub__(y)
	乘法	x * y	x.__mul__(y)
	除法	x / y	x.__truediv__(y)
	地板除	x // y	x.__floordiv__(y)
	取模（取余）	x % y	x.__mod__(y)
	地板除 & 取模	divmod(x, y)	x.__divmod__(y)
	乘幂	x ** y	x.__pow__(y)
	左位移	x << y	x.__lshift__(y)
	右位移	x >> y	x.__rshift__(y)
	按位 and	x & y	x.__and__(y)
	按位 xor	x ^ y	x.__xor__(y)
	按位 or	x	y	x.__or__(y)

如果 x 是某个实现了所有这些方法的类的实例，那么万事大吉。但如果未实现其中之一呢？或者更糟，如果实现了，但却无法处理某几类参数会怎么样？例如：

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> 1 / x
Fraction(3, 1)
```

这并 不是 传入一个 分数 并将其除以一个整数（如前例那样）的情况。前例中的情况非常直观： x / 3 调用 x.__truediv__(3) ，而 Fraction 的 __truediv__() 方法处理所有的数学运算。但整数并不“知道”如何对分数进行数学计算。因此本例该如何运作呢？

和 反映操作 相关的还有第二部分算数特殊方法。给定一个二元算术运算 （例如： x / y ），有两种方法来实现它：

1. 告诉 x 将自己除以 y ，或者
2. 告诉 y 去除 x

之前提到的特殊方法集合采用了第一种方式：对于给定 x / y ，它们为 x 提供了一种途径来表述“我知道如何将自己除以 y 。”下面的特殊方法集合采用了第二种方法：它们向 y 提供了一种途径来表述“我知道如何成为分母，并用自己去除 x 。”

序号	目的	所编写代码	Python 实际调用
	加法	x + y	y.__radd__(x)
	减法	x - y	y.__rsub__(x)
	乘法	x * y	y.__rmul__(x)
	除法	x / y	y.__rtruediv__(x)
	地板除	x // y	y.__rfloordiv__(x)
	取模（取余）	x % y	y.__rmod__(x)
	地板除 & 取模	divmod(x, y)	y.__rdivmod__(x)
	乘幂	x ** y	y.__rpow__(x)
	左位移	x << y	y.__rlshift__(x)
	右位移	x >> y	y.__rrshift__(x)
	按位 and	x & y	y.__rand__(x)
	按位 xor	x ^ y	y.__rxor__(x)
	按位 or	x	y	y.__ror__(x)

但是等一下！还有更多特殊方法！如果在进行“原地”操作，如： x /= 3 ，还可定义更多的特殊方法。

序号	目的	所编写代码	Python 实际调用
	原地加法	x += y	x.__iadd__(y)
	原地减法	x -= y	x.__isub__(y)
	原地乘法	x *= y	x.__imul__(y)
	原地除法	x /= y	x.__itruediv__(y)
	原地地板除法	x //= y	x.__ifloordiv__(y)
	原地取模	x %= y	x.__imod__(y)
	原地乘幂	x **= y	x.__ipow__(y)
	原地左位移	x <<= y	x.__ilshift__(y)
	原地右位移	x >>= y	x.__irshift__(y)
	原地按位 and	x &= y	x.__iand__(y)
	原地按位 xor	x ^= y	x.__ixor__(y)
	原地按位 or	x |= y	x.__ior__(y)

注意：多数情况下，并不需要原地操作方法。如果未对特定运算定义“就地”方法，Python 将会试着使用（普通）方法。例如，为执行表达式 `x /= y`，Python 将会：

1. 试着调用 `x.__itruediv__(` y `)`。如果该方法已经定义，并返回了 `NotImplemented` 之外的值，那已经大功告成了。
2. 试图调用 `x.__truediv__(` y `)`。如果该方法已定义并返回一个 `NotImplemented` 之外的值， `x` 的旧值将被丢弃，并将所返回的值替代它，就像是进行了 `x = x / y` 运算。
3. 试图调用 `y.__rtruediv__(` x `)`。如果该方法已定义并返回了一个 `NotImplemented` 之外的值， `x` 的旧值将被丢弃，并用所返回值进行替换。

因此如果想对原地运算进行优化，仅需像 `__itruediv__()` 方法一样定义“原地”方法。否则，基本上 Python 将会重新生成原地运算公式，以使用常规的运算及变量赋值。

还有一些“一元”数学运算，可以对“类-数字”对象自己执行。

序号	目的	所编写代码	Python 实际调用
	负数	`-x`	`x.__neg__()`
	正数	`+x`	`x.__pos__()`
	绝对值	`abs(x)`	`x.__abs__()`
	取反	`~x`	`x.__invert__()`
	复数	`complex(x)`	`x.__complex__()`
	整数转换	`int(x)`	`x.__int__()`
	浮点数	`float(x)`	`x.__float__()`
	四舍五入至最近的整数	`round(x)`	`x.__round__()`
	四舍五入至最近的 `n` 位小数	`round(x, n)`	`x.__round__(n)`
	`>= x` 的最小整数	`math.ceil(x)`	`x.__ceil__()`
	`<= x` 的最大整数	`math.floor(x)`	`x.__floor__()`
	对 `x` 朝向 0 取整	`math.trunc(x)`	`x.__trunc__()`
PEP 357	作为列表索引的数字	`a_list[x]`	`a_list[x.__index__()]`

可比较的类

我将此内容从前一节中拿出来使其单独成节，是因为“比较”操作并不局限于数字。许多数据类型都可以进行比较——字符串、列表，甚至字典。如果要创建自己的类，且对象之间的比较有意义，可以使用下面的特殊方法来实现比较。

序号	目的	所编写代码	Python 实际调用
	相等	`x == y`	`x.__eq__( y )`
	不相等	`x != y`	`x.__ne__( y )`
	小于	`x < y`	`x.__lt__( y )`
	小于或等于	`x <= y`	`x.__le__( y )`
	大于	`x > y`	`x.__gt__( y )`
	大于或等于	`x >= y`	`x.__ge__( y )`
	布尔上上下文环境中的真值	`if x:`	`x.__bool__()`

> ☞如果定义了 `__lt__()` 方法但没有定义 `__gt__()` 方法，Python 将通过经交换的算子调用 `__lt__()` 方法。然而，Python 并不会组合方法。例如，如果定义了 `__lt__()` 方法和 `__eq()__` 方法，并试图测试是否 `x <= y`，Python 不会按顺序调用 `__lt__()` 和 `__eq()__` 。它将只调用 `__le__()` 方法。

可序列化的类

Python 支持 任意对象的序列化和反序列化。（多数 Python 参考资料称该过程为 “pickling” 和 “unpickling”）。该技术对与将状态保存为文件并在稍后恢复它非常有意义。所有的 内置数据类型 均已支持 pickling 。如果创建了自定义类，且希望它能够 pickle，阅读 pickle 协议 了解下列特殊方法何时以及如何被调用。

序号	目的	所编写代码	Python 实际调
	自定义对象的复制	`copy.copy(x)`	`x.__copy__()`
	自定义对象的深度复制	`copy.deepcopy(x)`	`x.__deepcopy__()`
	在 pickling 之前获取对象的状态	`pickle.dump(x,` file `)`	`x.__getstate__()`
	序列化某对象	`pickle.dump(x,` file `)`	`x.__reduce__()`
	序列化某对象（新 pickling 协议）	`pickle.dump(x,` file `,` protocol_version `)`	`x.__reduce_ex__(` protoco
*	控制 unpickling 过程中对象的创建方式	`x = pickle.load(` file `)`	`x.__getnewargs__()`
*	在 unpickling 之后还原对象的状态	`x = pickle.load(` file `)`	`x.__setstate__()`

- 要重建序列化对象，Python 需要创建一个和被序列化的对象看起来一样的新对象，然后设置新对象的所有属性。`__getnewargs__()` 方法控制新对象的创建过程，而 `__setstate__()` 方法控制属性值的还原方式。

可在 `with` 语块中使用的类

`with` 语块定义了 运行时刻上下文环境；在执行 `with` 语句时将“进入”该上下文环境，而执行该语块中的最后一条语句将“退出”该上下文环境。

序号	目的	所编写代码	Python 实际调用
	在进入 `with` 语块时进行一些特别操作	`with x:`	`x.__enter__()`
	在退出 `with` 语块时进行一些特别操作	`with x:`	`x.__exit__()`

以下是 `with` file`` 习惯用法 的运作方式：

```
# excerpt from io.py:
def _checkClosed(self, msg=None):
    '''Internal: raise an ValueError if file is closed
    '''
    if self.closed:
        raise ValueError('I/O operation on closed file.'
                         if msg is None else msg)

def __enter__(self):
    '''Context management protocol.  Returns self.'''

def __exit__(self, *args):
    '''Context management protocol.  Calls close()'''
```

1. 该文件对象同时定义了一个 `__enter__()` 和一个 `__exit__()` 方法。该 `__enter__()` 方法检查文件是否处于打开状态；如果没有， `_checkClosed()` 方法引发一个例外。
2. `__enter__()` 方法将始终返回 `self` —— 这是 `with` 语块将用于调用属性和方法的对象
3. 在 `with` 语块结束后，文件对象将自动关闭。怎么做到的？在 `__exit__()` 方法中调用了 `self.close()` .

> ☞该 `__exit__()` 方法将总是被调用，哪怕是在 `with` 语块中引发了例外。实际上，如果引发了例外，该例外信息将会被传递给 `__exit__()` 方法。查阅 With 状态上下文环境管理器 了解更多细节。

要了解关于上下文管理器的更多内容，请查阅 《自动关闭文件》 和 《重定向标准输出》 。

真正神奇的东西

如果知道自己在干什么，你几乎可以完全控制类是如何比较的、属性如何定义，以及类的子类是何种类型。

序号	目的	所编写代码	Python 实际调用
	类构造器	`x = MyClass()`	`x.__new__()`
*	类析构器	`del x`	`x.__del__()`
	只定义特定集合的某些属性	`x.__slots__()`	
	自定义散列值	`hash(x)`	`x.__hash__()`
	获取		

	某个属性的值	`x.color`	`type(x).__dict__['color'].__get__(x, type(x))`
	设置某个属性的值	`x.color = 'PapayaWhip'`	`type(x).__dict__['color'].__set__(x, 'PapayaWh`
	删除某个属性	`del x.color`	`type(x).__dict__['color'].__del__(x)`
	控制某个对象是否是该对象的实例 your class	`isinstance(x, MyClass)`	`MyClass.__instancecheck__(x)`
	控制某个类是否是该类的子类	`issubclass(C, MyClass)`	`MyClass.__subclasscheck__(C)`
	控制某个类是否是该抽象基类的子类	`issubclass(C, MyABC)`	`MyABC.__subclasshook__(C)`

* 确切掌握 Python 何时调用 `__del__()` 特别方法是件难以置信的复杂事情。要想完全理解它，必须清楚 Python 如何在内存中跟踪对象。以下有一篇好文章介绍 Python 垃圾收集和类析构器。还可以阅读《弱引用》、《 `weakref` 模块》，还可以将《 `gc` 模块》当作补充阅读材料。

深入阅读

本附录中提到的模块：

- `zipfile` 模块

- `cgi` 模块
- `collections` 模块
- `math [数学]` 模块
- `pickle` 模块
- `copy` 模块
- `abc` (“抽象基类”) 模块

其它启发式阅读：

- 迷你语言格式规范
- Python 数据模型
- 内建类型
- PEP 357: 使任何对象可以使用切片
- PEP 3119: 抽象基类简介

Chapter C 接下来阅读什么？

> " Go forth on your path, as it exists only through your walking. " — St. Augustine of Hippo (attributed)

要阅读的对象

鉴于一些主题有免费的教程，因此我决定不在本书中加以阐述。

修饰器：

- 函数修饰器 作者： Ariel Ortiz
- 关于函数修饰器的更多讨论 作者： Ariel Ortiz
- 可爱的 Python：修饰器使魔法更轻松 作者： David Mertz
- 官方 Python 文档中的 函数定义

属性：

- Python 内建 `属性` 作者： Adam Gomaa
- Getters/Setters/Fuxors 作者： Ryan Tomayko
- 官方 Python 文档中的 `property()` 函数

描述符：

- 描述符的 How-To 指南 作者: Raymond Hettinger
- 可爱的 Python: Python 的简洁与累赘，第二部分 作者： David Mertz
- Python 描述符 作者： Mark Summerfield
- Python 官方文档中的 调用描述符

线程 & 多进程:

- `threading` 模块
- `线程` — 管理并发线程
- `multiprocessing` 模块
- `多进程` — 像管理线程那样管理进程
- Python 线程和全局解释器锁 作者： Jesse Noller
- Python GIL 揭密（视频） 作者： David Beazley

元类

- Python 中的元类编程 作者： David Mertz 和 Michele Simionato
- Python 中的元类编程，第二部分 作者： David Mertz 和 Michele Simionato
- Python 中的元类编程，第三部分 作者： David Mertz 和 Michele Simionato

此外，Doug Hellman 之 本周 Python 模块 是对Python 标准类库模块的极好指南

到哪里找与 Python 3-兼容的代码

由于 Python 3 相对较新，其非常缺乏兼容类库。以下地方可用于查找在 Python 3 之下能够正常运作的代码：

- Python 安装包索引：Python 3 安装包清单
- Python 食谱：标记了 “python3”的内容清单
- 以 Google 为宿主的项目：标记为 “python3”的项目清单
- SourceForge: 符合 “Python 3”的项目清单
- GitHub: 符合 “python3”的项目清单 (以及 符合 “python 3”的项目清单)
- BitBucket: 符合 “python3”的项目清单 (以及 符合 “python 3”的项目清单)